Answer Set Programming: From Theory to Practice

Roland Kaminski Javier Romero Torsten Schaub Philipp Wanko

University of Potsdam

Outline

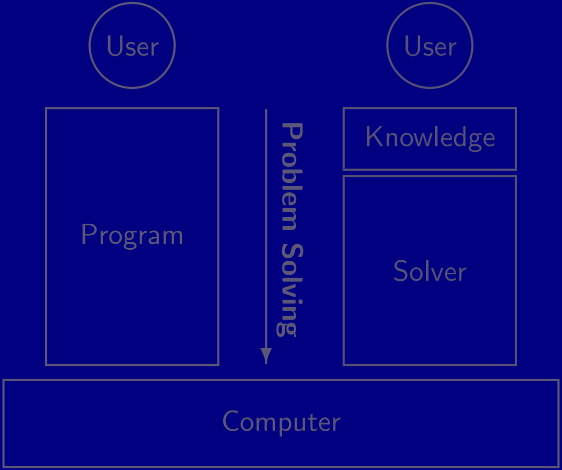
Full version `https://teaching.potassco.org`

Potassco

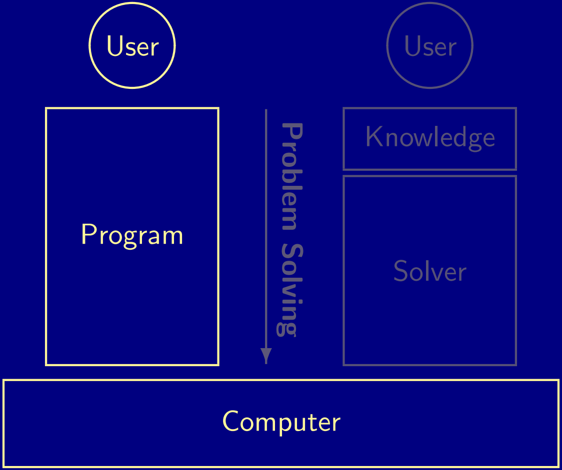
Introduction: Overview

Outline

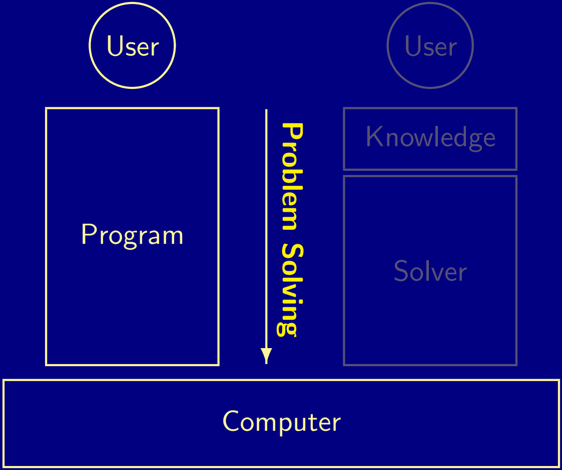
Traditional Software



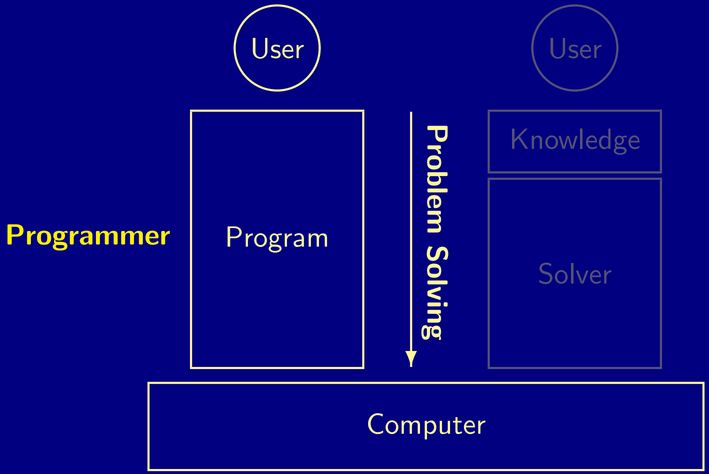
Traditional Software



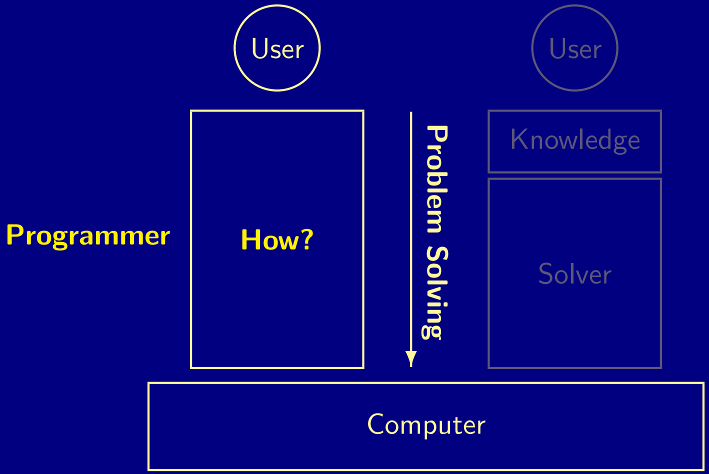
Traditional Software



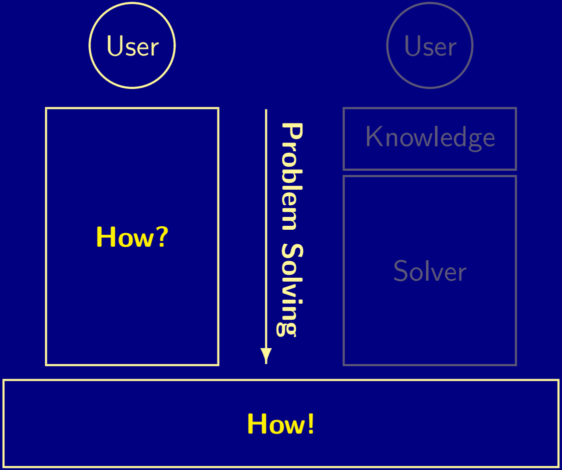
Traditional Software



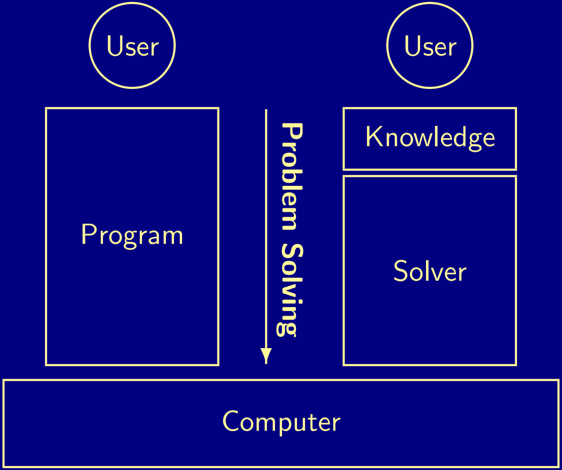
Traditional Software



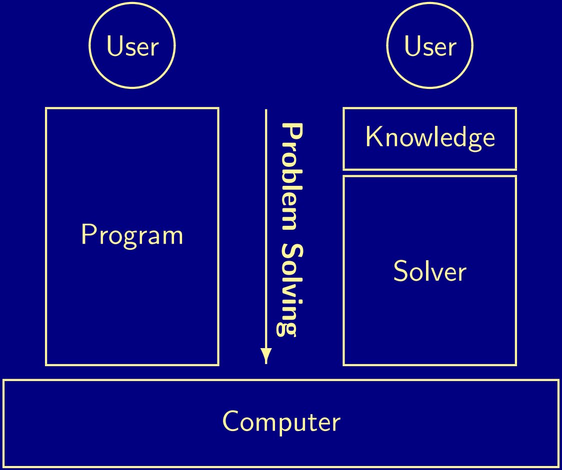
Traditional Software



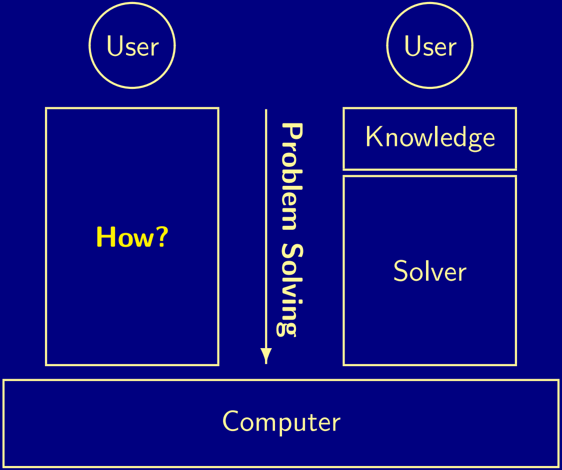
Declarative Software



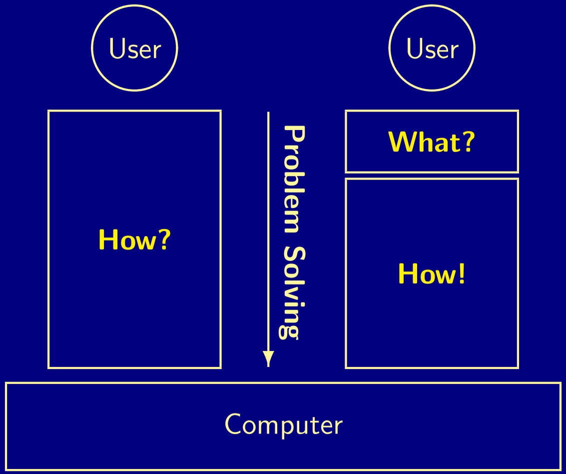
Knowledge-driven Software



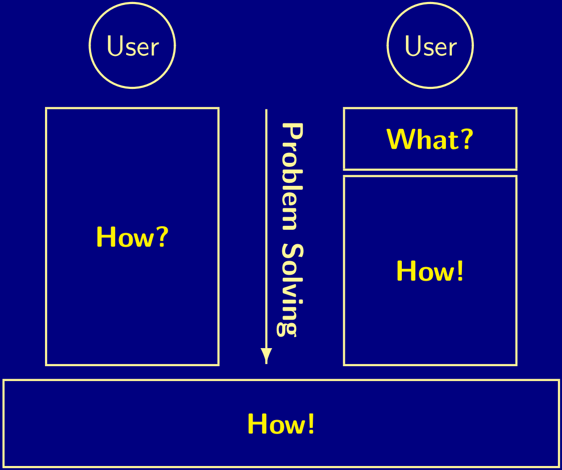
Knowledge-driven Software



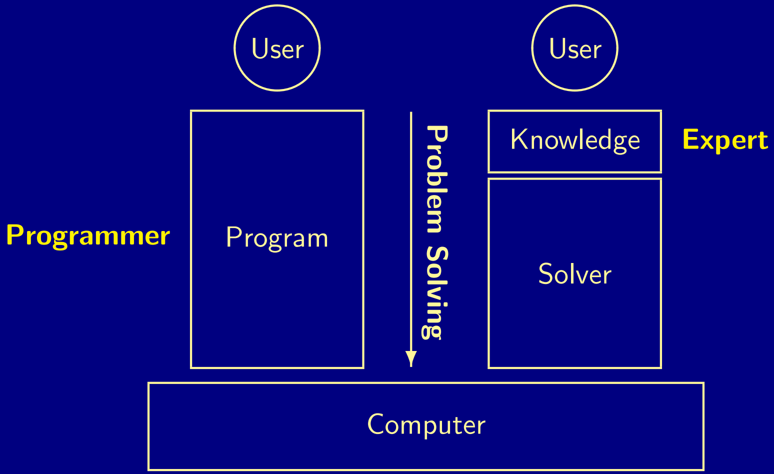
Knowledge-driven Software



Knowledge-driven Software



Knowledge-driven Software



What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge **Expert**

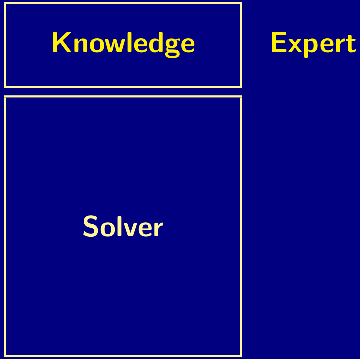
Solver

Potassco

What is the benefit?

- \+ Transparency
- \+ Flexibility
- \+ Maintainability
- \+ Reliability

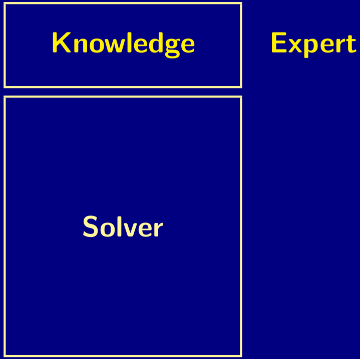
- \+ Generality
- \+ Efficiency
- \+ Optimality
- \+ Availability



What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

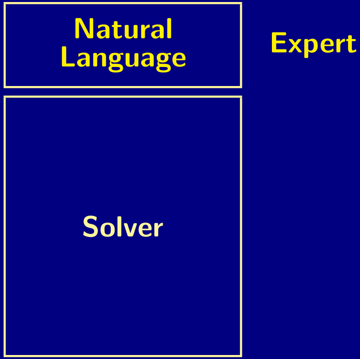
+ Generality
+ Efficiency
+ Optimality
+ Availability



What is the benefit?

- + Transparency
- + Flexibility
- + Maintainability
- + Reliability

- + Generality
- + Efficiency
- + Optimality
- + Availability



What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Natural Language | **Layperson**

Solver

What is the benefit?

- + Transparency
- + Flexibility
- + Maintainability
- + Reliability

- + Generality
- + Efficiency
- + Optimality
- + Availability

Knowledge **Expert**

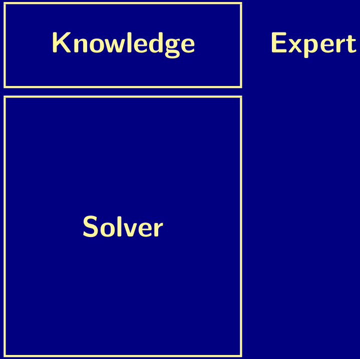
Solver

Potassco

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability



What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge **Expert**

Solver

Potassco

Outline

1 Motivation

2 Nutshell

3 Evolution

4 Workflow

5 Usage

Answer Set Programming (ASP)

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

Answer Set Programming (ASP)

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

Answer Set Programming (ASP)

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

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

Answer Set Programming (ASP)

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

Answer Set Programming (ASP)

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

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

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
  - Siemens: Partner units configuration
  - Variantum: Product configuration
  - US Navy: risk assessment

Answer Set Programming (ASP)

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

Answer Set Programming (ASP)

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

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

Outline

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

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

Some (biased) moments in time

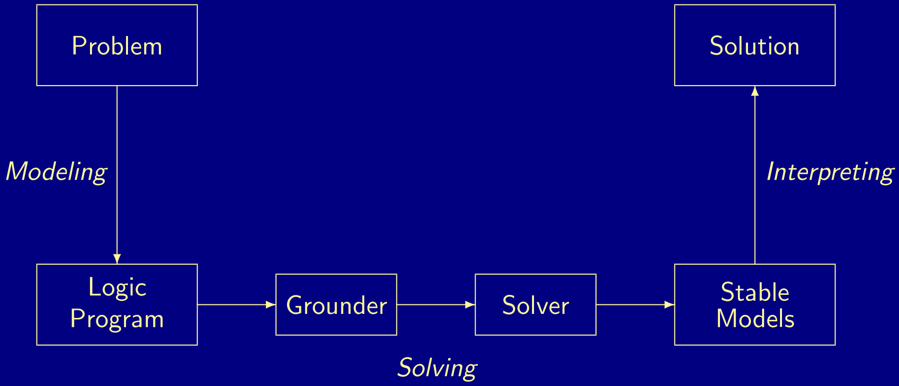
- '80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models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

Some (biased) moments in time

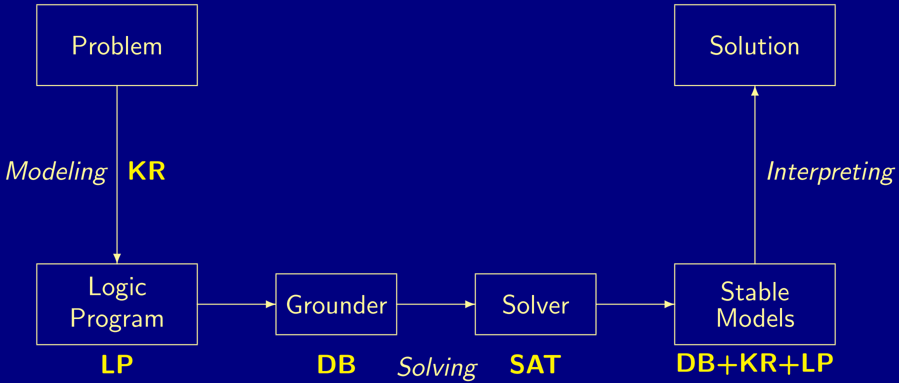
- '80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models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

Outline

Modeling, grounding, and solving



Rooting ASP


Problem
Solution
Modeling
KR
Interpreting
Logic Program
Grounder
Solver
Stable Models
LP
DB
Solving
SAT
DB+KR+LP

Outline

Two sides of a coin

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

Two sides of a coin

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

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

Two sides of a coin

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

Two and a half sides of a coin

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

Foundations: Overview

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, [4, 5]

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, [4, 5]

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, [4, 5]

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, [4, 5]

Propositional Normal Logic Programs

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

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

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

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

Potassco

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

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

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

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

Logic Programs

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

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

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

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

Normal Logic Programs

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

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

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

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

- Disclaimer The following formalities apply to normal logic programs

Potassco

Outline

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

Potassco

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

Some truth tabling, back to SAT

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

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

Some truth tabling, and now ASP

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

Some truth tabling, and now ASP

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

Some truth tabling, and now ASP

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

Some truth tabling, and now ASP

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

Some truth tabling, and now ASP

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

Potassco

Some truth tabling, and now ASP

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

Potassco

Some truth tabling, and now ASP

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

Reduct

Potassco

Some truth tabling, and now ASP

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

Reduct

Some truth tabling, and now ASP

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

Reduct

Some truth tabling, and now ASP

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

Reduct

Potassco

Some truth tabling, and now ASP

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

Reduct

Potassco

Some truth tabling, and now ASP

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

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

Potassco

Some truth tabling, and now ASP

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

Reduct

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

Stable model

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

$$a \leftarrow b_1, \dots, b_m, \neg c_1, \dots, \neg c_n$$

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

$$P^X = \{a \leftarrow b_1, \dots, b_m \mid r \in P,\ \{c_1, \dots, c_n\} \cap X = \emptyset\}$$

- $Cn(P)$ stands for the smallest model of a positive program P

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

Potassco

Stable model

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

$$a \leftarrow b_1, \dots, b_m, \neg c_1, \dots, \neg c_n$$

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

$$P^X = \{a \leftarrow b_1, \dots, b_m \mid r \in P, \ \{c_1, \dots, c_n\} \cap X = \emptyset\}$$

- $Cn(P)$ stands for the smallest model of a positive program P

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

Stable model

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

$$a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n$$

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

$$P^X = \{a \leftarrow b_1, \ldots, b_m \mid r \in P,\ \{c_1, \ldots, c_n\} \cap X = \emptyset\}$$

- $Cn(P)$ stands for the smallest model of a positive program P

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

Outline

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$CF(P) = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow (a \wedge \neg c) \vee y \quad y \leftarrow x \wedge b\}$$
$$\cup \{c \leftrightarrow \bot\}$$
$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $CF(P)$:

$\{b\}$, $\{b,c\}$, $\{b,x,y\}$, $\{b,c,x,y\}$, $\{a,c\}$, $\{a,b,c\}$, $\{a,x\}$, $\{a,c,x\}$, $\{a,x,y\}$, $\{a,c,x,y\}$, $\{a,b,x,y\}$, $\{a,b,c,x,y\}$

Unsupported atoms

Unfounded atoms

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$RF(P) = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow (a \wedge \neg c) \vee y \quad y \leftarrow x \wedge b\}$$
$$\cup \{c \leftrightarrow \bot\}$$

$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $RF(P)$: (only true atoms shown)

$\{b\}$, $\{b, c\}$, $\{b, x, y\}$, $\{b, c, x, y\}$, $\{a, c\}$, $\{a, b, c\}$, $\{a, x\}$, $\{a, c, x\}$, $\{a, x, y\}$, $\{a, c, x, y\}$, $\{a, b, x, y\}$, $\{a, b, c, x, y\}$

- Unsupported atoms
- Unfounded atoms

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$RF(P) = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow (a \wedge \neg c) \vee y \quad y \leftarrow x \wedge b\}$$

$$\cup \{c \leftrightarrow \bot\}$$

$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $RF(P)$:

$\{b\}$, $\{b, c\}$, $\{b, x, y\}$, $\{b, c, x, y\}$, $\{a, c\}$, $\{a, b, c\}$, $\{a, x\}$, $\{a, c, x\}$, $\{a, x, y\}$, $\{a, c, x, y\}$, $\{a, b, x, y\}$, $\{a, b, c, x, y\}$

- Unsupported atoms
- Unfounded atoms

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$CF(P) = \{a \leftrightarrow \neg b \quad b \leftrightarrow \neg a \quad x \leftrightarrow (a \wedge \neg c) \vee y \quad y \leftrightarrow x \wedge b\}$$
$$\cup \{c \leftrightarrow \bot\}$$

$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $RF(P)$:

$\{b\}$, $\{b, c\}$, $\{b, x, y\}$, $\{b, c, x, y\}$, $\{a, c\}$, $\{a, b, c\}$, $\{a, x\}$, $\{a, c, x\}$, $\{a, x, y\}$, $\{a, c, x, y\}$, $\{a, b, x, y\}$, $\{a, b, c, x, y\}$

- Unsupported atoms
- Unfounded atoms

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$CF(P) = \{a \leftrightarrow \neg b \quad b \leftrightarrow \neg a \quad x \leftrightarrow (a \wedge \neg c) \vee y \quad y \leftrightarrow x \wedge b\}$$
$$\cup \{c \leftrightarrow \bot\}$$

$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $CF(P)$:

$\{b\}$, $\{b, c\}$, $\{b, x, y\}$, $\{b, c, x, y\}$, $\{a, c\}$, $\{a, b, c\}$, $\{a, x\}$, $\{a, c, x\}$, $\{a, x, y\}$, $\{a, c, x, y\}$, $\{a, b, x, y\}$, $\{a, b, c, x, y\}$

- ~~Unsupported atoms~~
- Unfounded atoms

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$CF(P) = \{a \leftrightarrow \neg b \quad b \leftrightarrow \neg a \quad x \leftrightarrow (a \wedge \neg c) \vee y \quad y \leftrightarrow x \wedge b\}$$
$$\cup \{c \leftrightarrow \bot\}$$

$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $CF(P)$:

$\{b\}$, $\{b, c\}$, $\{b, x, y\}$, $\{b, c, x, y\}$, $\{a, c\}$, $\{a, b, c\}$, $\{a, x\}$, $\{a, c, x\}$, $\{a, x, y\}$, $\{a, c, x, y\}$, $\{a, b, x, y\}$, $\{a, b, c, x, y\}$

- ~~Unsupported atoms~~
- Unfounded atoms

Potassco

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$CF(P) = \{a \leftrightarrow \neg b \quad b \leftrightarrow \neg a \quad x \leftrightarrow (a \wedge \neg c) \vee y \quad y \leftrightarrow x \wedge b\}$$
$$\cup \{c \leftrightarrow \bot\}$$

$$LF(P) = \{(x \vee y) \rightarrow a \wedge \neg c\}$$

Classical models of $CF(P) \cup LF(P)$:

$\{b\}$, $\{b, c\}$, $\{b, x, y\}$, $\{b, c, x, y\}$, $\{a, c\}$, $\{a, b, c\}$, $\{a, x\}$, $\{a, c, x\}$, $\{a, x, y\}$, $\{a, c, x, y\}$, $\{a, b, x, y\}$, $\{a, b, c, x, y\}$

- ~~Unsupported atoms~~
- ~~Unfounded atoms~~

Logic Programs as Propositional Formulas

$$P = \{a \leftarrow \neg b \quad b \leftarrow \neg a \quad x \leftarrow a, \neg c \quad x \leftarrow y \quad y \leftarrow x, b\}$$

$$CF(P) = \{a \leftrightarrow (\bigvee_{(a \leftarrow B) \in P} BF(B)) \mid a \in A(P)\}$$

$$BF(B) = \bigwedge_{b \in B \cap A(P)} b \wedge \bigwedge_{\neg c \in B} \neg c$$

$$LF(P) = \{(\bigvee_{a \in L} a) \rightarrow (\bigvee_{a \in L, (a \leftarrow B) \in P, B \cap L = \emptyset} BF(B)) \mid L \in loop(P)\}$$

Classical models of $CF(P) \cup LF(P)$:

Theorem (Lin and Zhao)

Let P be a normal logic program and $X \subseteq A(P)$.
Then, X is a stable model of P iff $X \models CF(P) \cup LF(P)$.

- Size of $CF(P)$ is linear in the size of P
- Size of $LF(P)$ may be exponential in the size of P

Potassco

ASP and SAT

- SAT = ASP + Law of the excluded middle
- ASP = SAT + Completion and Loop formulas

- Note Checking whether a propositional formula has a stable model is Σ_P^2-complete

ASP and SAT

- SAT = ASP + Law of the excluded middle
- ASP = SAT + Completion and Loop formulas

- Note Checking whether a propositional formula has a stable model is Σ_P^2-complete

Potassco

ASP and SAT

- SAT = ASP + Law of the excluded middle[1]
- ASP = SAT + Completion and Loop formulas

- Note Checking whether a propositional formula has a stable model is Σ^2_P-complete

[1]For instance, '{a}.' stands for '$a \vee \neg a$'.

ASP and SAT

- SAT = ASP + Law of the excluded middle
- ASP = SAT + Completion and Loop formulas

- Note Checking whether a propositional formula has a stable model is Σ_P^2-complete

Outline

The logic of Here-and-There (HT)

- An interpretation is a pair $\langle H, T \rangle$ of sets of atoms with $H \subseteq T$
  - H is called "here" and
  - T is called "there"
- Note $\langle H, T \rangle$ is a simplified Kripke structure
- Intuition
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Idea
  - $\langle H, T \rangle \models \varphi \quad \sim \quad \varphi$ is provably true
  - $\langle T, T \rangle \models \varphi \quad \sim \quad \varphi$ is possibly true, that is, classically true

Potassco

The logic of Here-and-There (HT)

- An interpretation is a pair $\langle H, T \rangle$ of sets of atoms with $H \subseteq T$
  - H is called "here" and
  - T is called "there"
- Note $\langle H, T \rangle$ is a simplified Kripke structure
- Intuition
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Idea
  - $\langle H, T \rangle \models \varphi \quad \sim \quad \varphi$ is provably true
  - $\langle T, T \rangle \models \varphi \quad \sim \quad \varphi$ is possibly true, that is, classically true

Potassco

The logic of Here-and-There (HT)

- An interpretation is a pair $\langle H, T\rangle$ of sets of atoms with $H \subseteq T$
  - H is called "here" and
  - T is called "there"
- Note $\langle H, T\rangle$ is a simplified Kripke structure
- Intuition
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Idea
  - $\langle H, T\rangle \models \varphi \quad \sim \quad \varphi$ is provably true
  - $\langle T, T\rangle \models \varphi \quad \sim \quad \varphi$ is possibly true, that is, classically true

Potassco

The logic of Here-and-There (HT)

- An interpretation is a pair $\langle H, T\rangle$ of sets of atoms with $H \subseteq T$
  - H is called "here" and
  - T is called "there"
- Note $\langle H, T\rangle$ is a simplified Kripke structure
- Intuition
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Idea
  - $\langle H, T\rangle \models \varphi \quad \sim \quad \varphi$ is provably true
  - $\langle T, T\rangle \models \varphi \quad \sim \quad \varphi$ is possibly true, that is, classically true

Potassco

The logic of Here-and-There (HT)

- An interpretation is a pair $\langle H, T\rangle$ of sets of atoms with $H \subseteq T$
  - H is called "here" and
  - T is called "there"
- Note $\langle H, T\rangle$ is a simplified Kripke structure
- Intuition
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Idea
  - $\langle H, T\rangle \models \varphi \quad \sim \quad \varphi$ is provably true
  - $\langle T, T\rangle \models \varphi \quad \sim \quad \varphi$ is possibly true, that is, classically true

Satisfaction

- $\langle H, T \rangle \models a$ if $a \in H$ for any atom a
- $\langle H, T \rangle \models \varphi \wedge \psi$ if $\langle H, T \rangle \models \varphi$ and $\langle H, T \rangle \models \psi$
- $\langle H, T \rangle \models \varphi \vee \psi$ if $\langle H, T \rangle \models \varphi$ or $\langle H, T \rangle \models \psi$
- $\langle H, T \rangle \models \varphi \rightarrow \psi$ if $\langle X, T \rangle \models \varphi$ implies $\langle X, T \rangle \models \psi$ for both $X = H, T$

- Note $\langle H, T \rangle \models \neg\varphi$ if $\langle T, T \rangle \not\models \varphi$ since $\neg\varphi = \varphi \rightarrow \bot$

- An interpretation $\langle H, T \rangle$ is a model of φ, if $\langle H, T \rangle \models \varphi$

Potassco

Satisfaction

- $\langle H, T \rangle \models a$ if $a \in H$ for any atom a
- $\langle H, T \rangle \models \varphi \wedge \psi$ if $\langle H, T \rangle \models \varphi$ and $\langle H, T \rangle \models \psi$
- $\langle H, T \rangle \models \varphi \vee \psi$ if $\langle H, T \rangle \models \varphi$ or $\langle H, T \rangle \models \psi$
- $\langle H, T \rangle \models \varphi \rightarrow \psi$ if $\langle X, T \rangle \models \varphi$ implies $\langle X, T \rangle \models \psi$ for both $X = H, T$
- Note $\langle H, T \rangle \models \neg\varphi$ if $\langle T, T \rangle \not\models \varphi$ since $\neg\varphi = \varphi \rightarrow \bot$
- An interpretation $\langle H, T \rangle$ is a model of φ, if $\langle H, T \rangle \models \varphi$

Satisfaction

- $\langle H, T \rangle \models a$ if $a \in H$ for any atom a
- $\langle H, T \rangle \models \varphi \wedge \psi$ if $\langle H, T \rangle \models \varphi$ and $\langle H, T \rangle \models \psi$
- $\langle H, T \rangle \models \varphi \vee \psi$ if $\langle H, T \rangle \models \varphi$ or $\langle H, T \rangle \models \psi$
- $\langle H, T \rangle \models \varphi \rightarrow \psi$ if $\langle X, T \rangle \models \varphi$ implies $\langle X, T \rangle \models \psi$ for both $X = H, T$

- Note $\langle H, T \rangle \models \neg\varphi$ if $\langle T, T \rangle \not\models \varphi$ since $\neg\varphi = \varphi \rightarrow \bot$

- An interpretation $\langle H, T \rangle$ is a model of φ, if $\langle H, T \rangle \models \varphi$

Classical tautologies

H	T	a	$\neg a$	$a \vee \neg a$	$\neg\neg a$	$a \leftarrow \neg\neg a$
$\{a\}$	$\{a\}$	***T***	***F***	***T***	***T***	***T***
$\emptyset$	$\{a\}$	***F***	***F***	***F***	***T***	***F***
$\emptyset$	$\emptyset$	***F***	***T***	***T***	***F***	***T***

Equilibrium models

- A total interpretation $\langle T, T \rangle$ is an equilibrium model of a formula φ, if
  1. $\langle T, T \rangle \models \varphi$,
  2. $\langle H, T \rangle \not\models \varphi$ for all $H \subset T$
- T is called a stable model of φ
- Note $\langle T, T \rangle$ acts as a classical model
- Note $\langle H, T \rangle \models P$ iff $H \models P^T$ (P^T is the reduct of P by T)

Equilibrium models

- A total interpretation $\langle T, T\rangle$ is an equilibrium model of a formula φ, if
  1. $\langle T, T\rangle \models \varphi$,
  2. $\langle H, T\rangle \not\models \varphi$ for all $H \subset T$
- T is called a stable model of φ
- Note $\langle T, T\rangle$ acts as a classical model
- Note $\langle H, T\rangle \models P$ iff $H \models P^T$ (P^T is the reduct of P by T)

Equilibrium models

- A total interpretation $\langle T, T\rangle$ is an equilibrium model of a formula φ, if
  1. $\langle T, T\rangle \models \varphi$,
  2. $\langle H, T\rangle \not\models \varphi$ for all $H \subset T$
- T is called a stable model of φ
- Note $\langle T, T\rangle$ acts as a classical model
- Note $\langle H, T\rangle \models P$ iff $H \models P^T$ (P^T is the reduct of P by T)

Potassco

Equilibrium models

- A total interpretation $\langle T, T \rangle$ is an equilibrium model of a formula φ, if
  1. $\langle T, T \rangle \models \varphi$,
  2. $\langle H, T \rangle \not\models \varphi$ for all $H \subset T$
- T is called a stable model of φ

- Note $\langle T, T \rangle$ acts as a classical model
- Note $\langle H, T \rangle \models P$ iff $H \models P^T$ (P^T is the reduct of P by T)

Grounding: Overview

Potassco

Outline

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 $\mathcal{T}$:

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

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 $\mathcal{T}$:

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

  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

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

Potassco

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 $\mathcal{T}$:

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

  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

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

Potassco

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 $\mathcal{T}$:

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

  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

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

Potassco

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 $\mathcal{T}$:

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

  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 $\mathcal{T}$:

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

  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 $\mathcal{T}$:

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

  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

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

Potassco

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 $\mathcal{T}$:

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

  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 $\mathcal{T}$:

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

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

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

Potassco

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

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

Potassco

An example

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

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

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

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

Grounding aims at reducing the ground instantiation
by applying semantic principles

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

by applying semantic principles

Potassco

An example

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

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

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

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

Grounding aims at reducing the ground instantiation
by applying semantic principles

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

by applying semantic principles

Potassco

An example

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

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

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

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

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

An example

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

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

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

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

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

Potassco

An example

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

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

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

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

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

An example

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

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

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

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

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

Potassco

Outline

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 $ground(P)$

Outline

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

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

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

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

     Rules: $\{ r(a,b) \leftarrow ,\ r(b,c) \leftarrow \}$
     Atoms: $\{ r(a,b), r(b,c) \}$
  3. 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: $\{ r(a,b), r(b,c), t(a,b), t(b,c) \}$
  4. Resulting ground rules

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

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

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

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

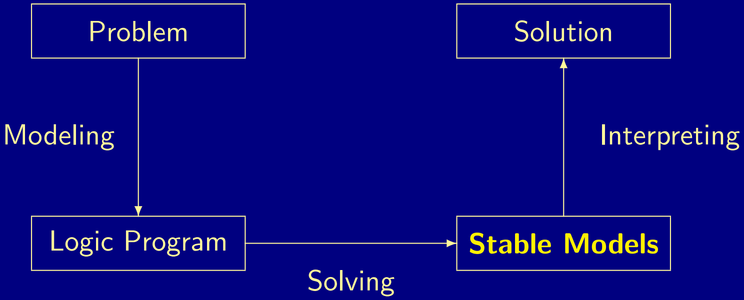
Grounding safe programs

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

Solving: Overview

Reasoning modes



Reasoning modes

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

- and combinations of them

[†] without solution recording

[‡] without solution enumeration

Outline

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
```

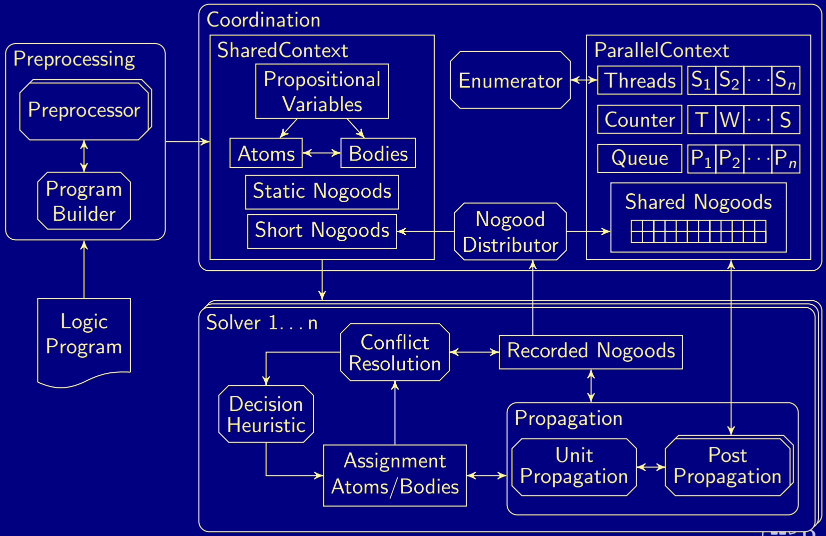
CDCL-style solving

loop

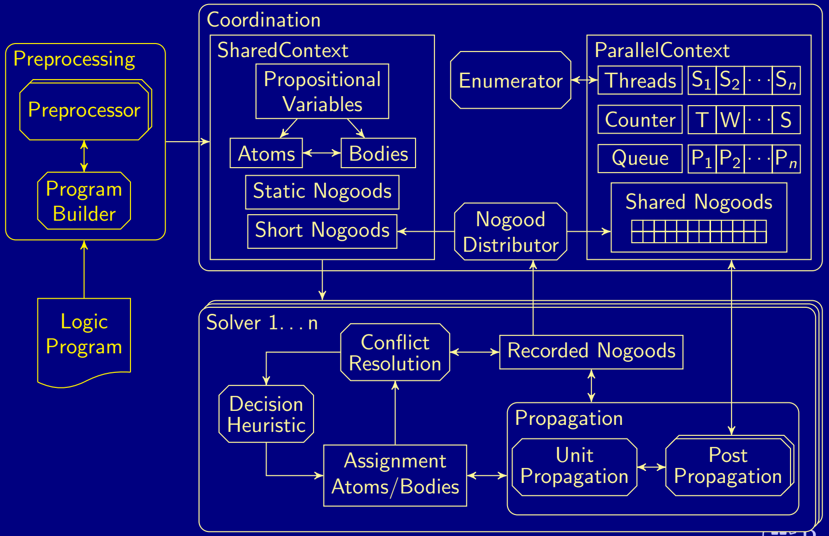
    propagate // deterministically assign literals
    if no conflict **then**
        if all variables assigned **then return** solution
        else *decide* // non-deterministically assign some literal
    else
        if top-level conflict **then return** unsatisfiable
        else
            analyze // analyze conflict and add conflict constraint
            backjump // unassign literals until conflict constraint is unit

Outline

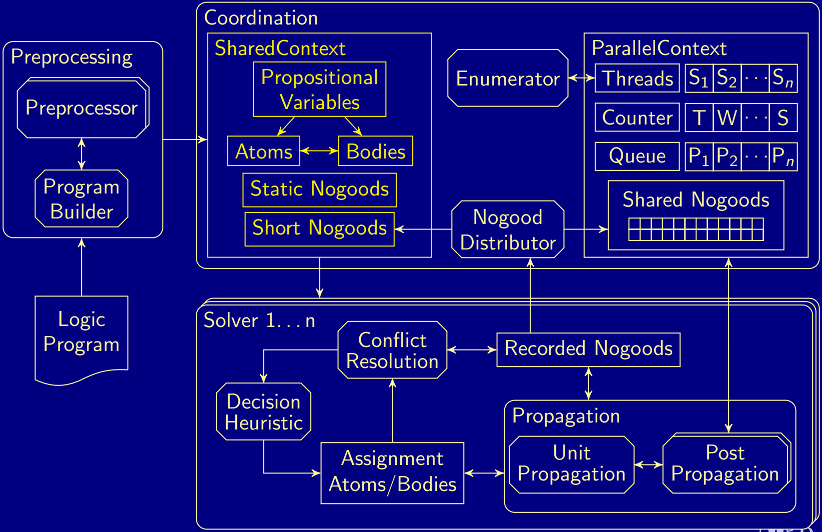
Multi-threaded architecture of *clasp*



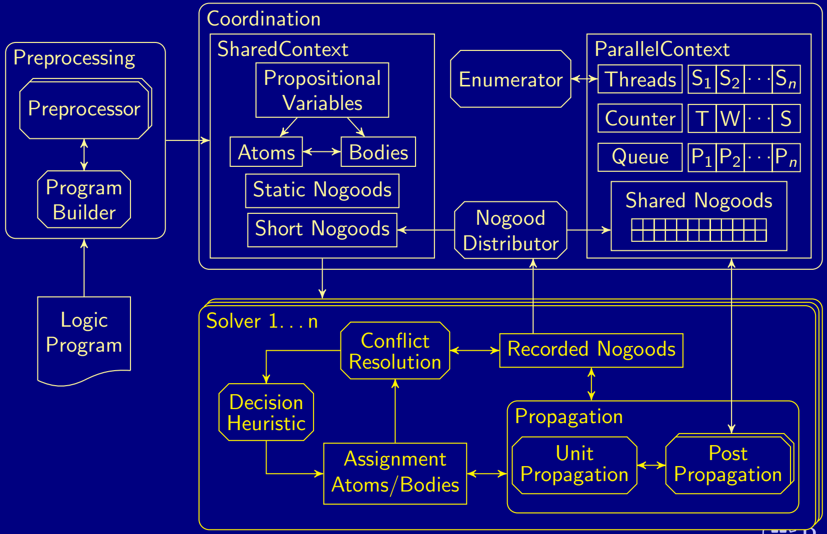
Multi-threaded architecture of *clasp*



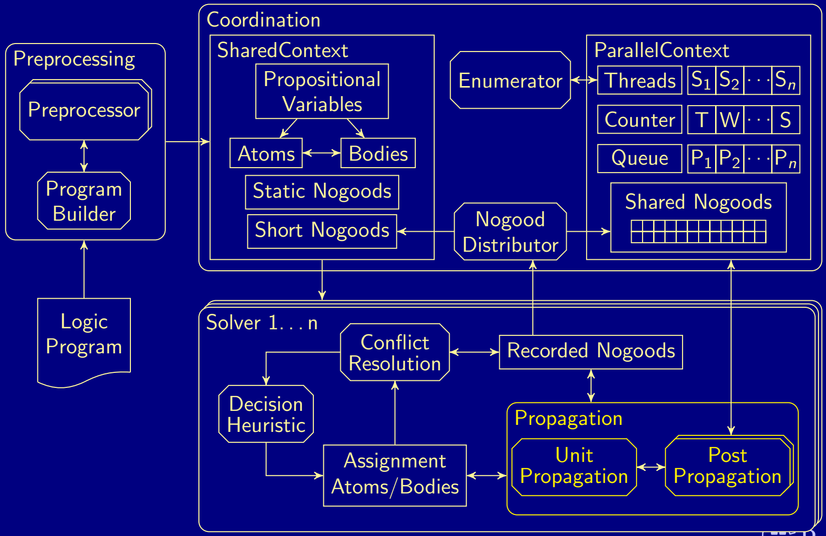
Multi-threaded architecture of *clasp*



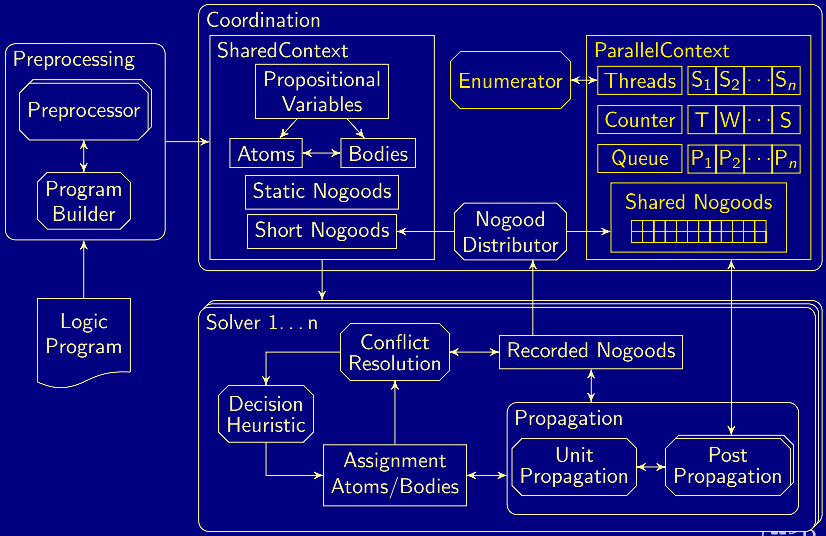
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*

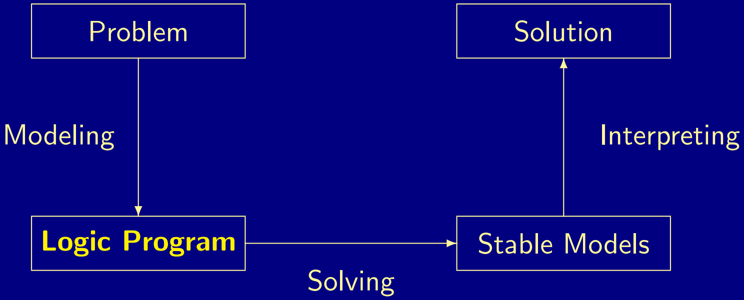


Multi-threaded architecture of *clasp*



Modeling: Overview

Extended syntax



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) }`

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) }`

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) }`

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) }`

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

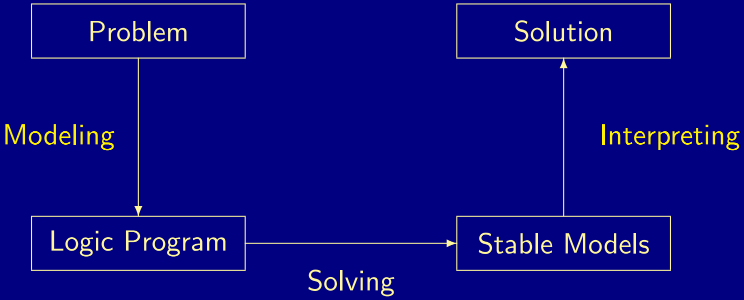
Language constructs

- 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

Modeling and Interpreting



Outline

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**

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."*

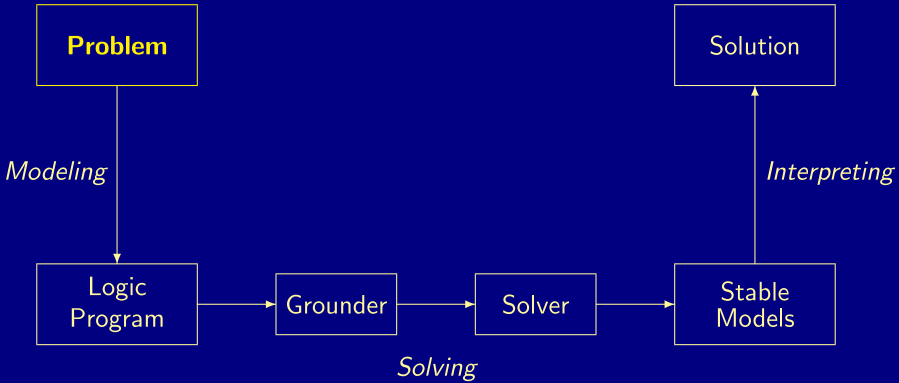
- Uniform problem representation

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

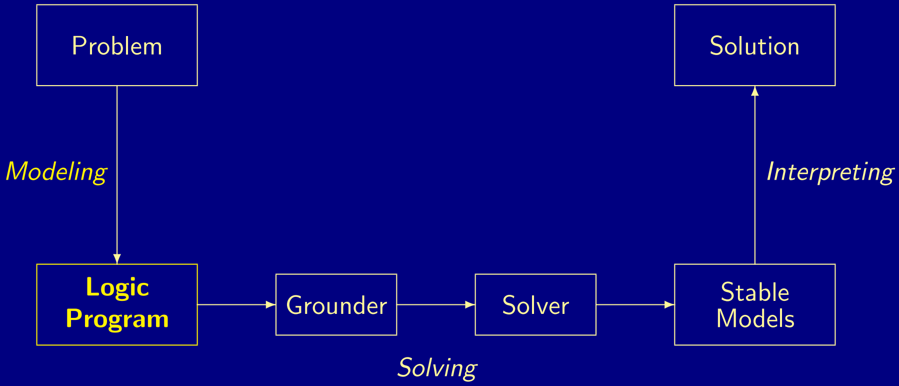
  - **I** is represented as a set of facts $P_{\mathbf{I}}$,
  - **C** is represented as a set of rules $P_{\mathbf{C}}$, and
  - $P_{\mathbf{C}}$ can be used to solve all problem instances in **C**

Outline

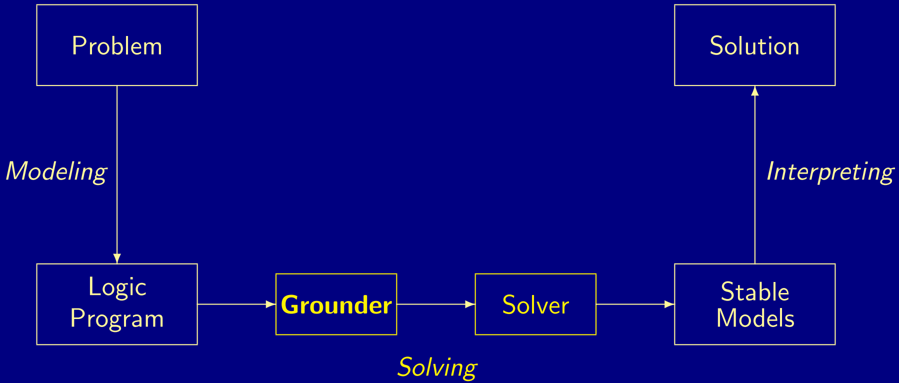
ASP workflow



ASP workflow

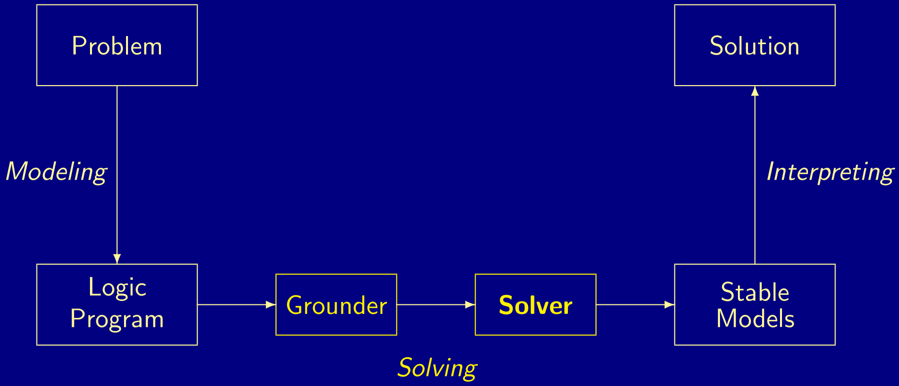


ASP workflow

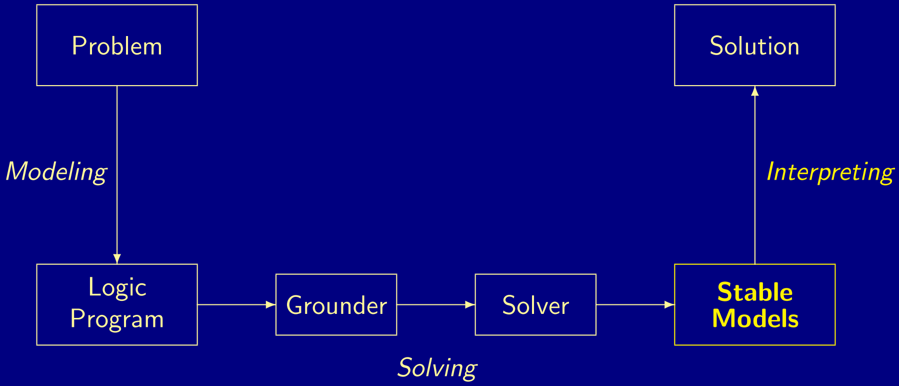


Potassco

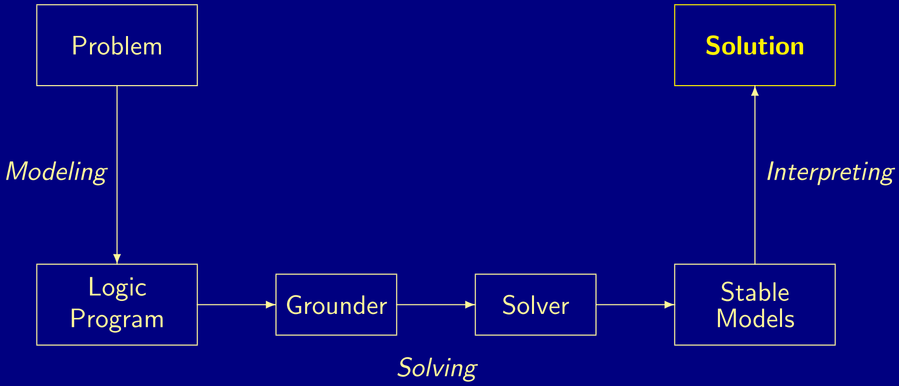
ASP workflow



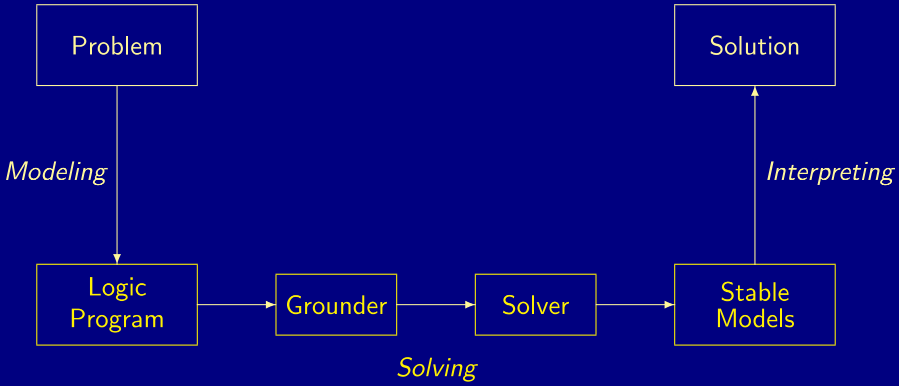
ASP workflow



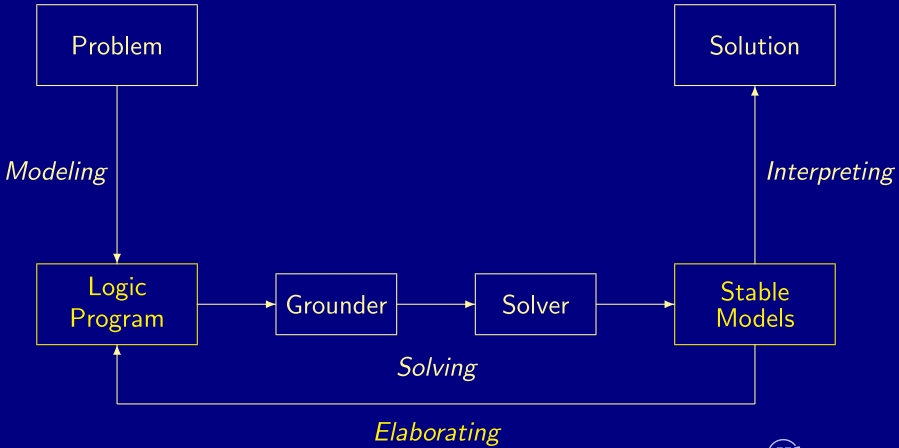
ASP workflow



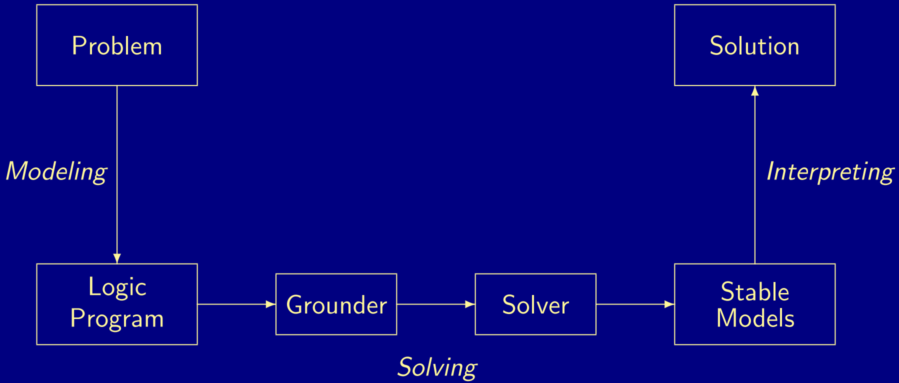
ASP workflow



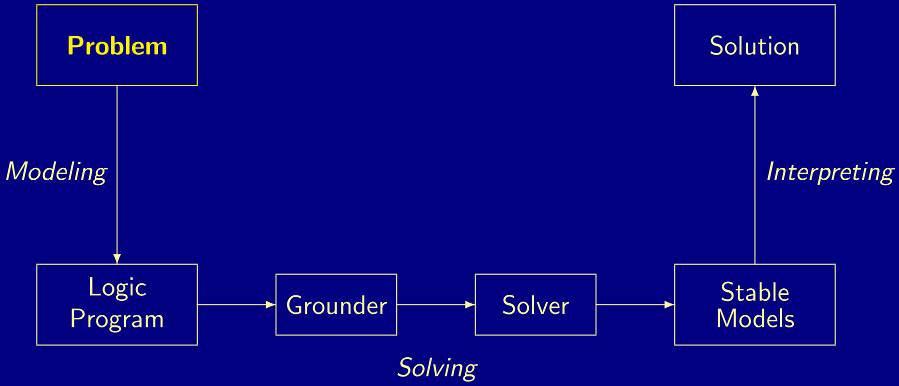
ASP workflow



ASP workflow



ASP workflow: Problem



A case-study: Graph coloring

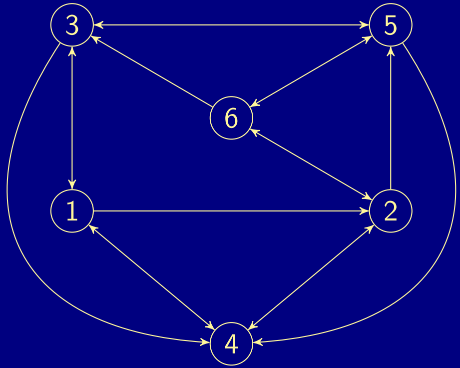
- Problem instance A graph consisting of nodes and edges

A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges

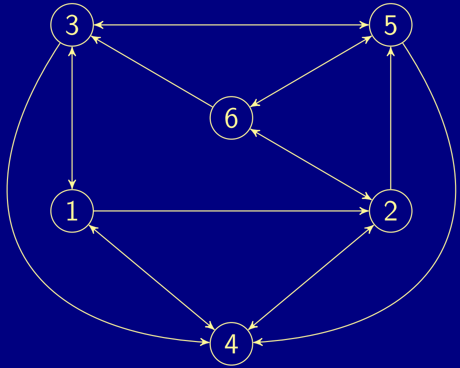
A case-study: Graph coloring

- 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

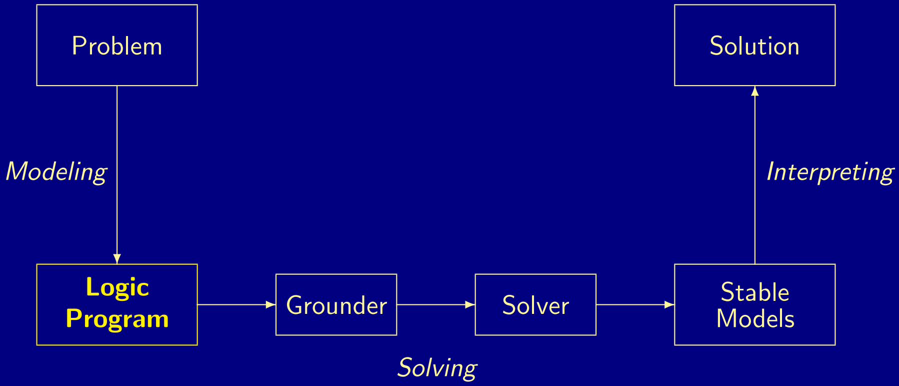
A case-study: Graph coloring

- 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

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

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

} Problem instance

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

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

} Problem encoding

Graph coloring

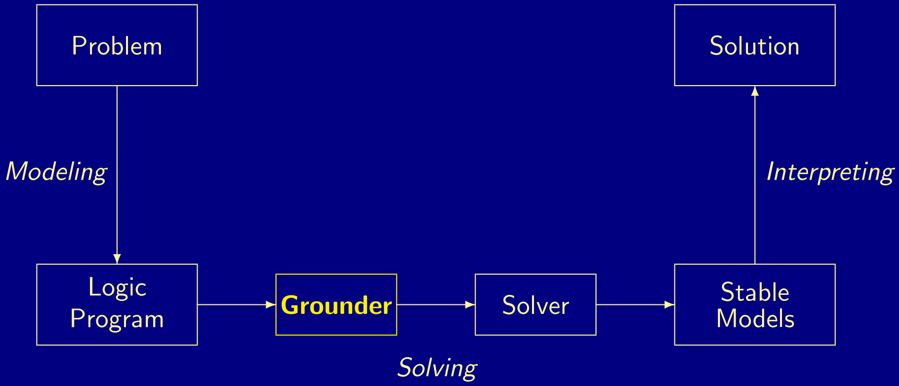
graph.lp:

```
node(1..6).

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

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

color.lp:

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

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

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).
```

Graph coloring: Grounding

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

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

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

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

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

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

Graph coloring: Grounding

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

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

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

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

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

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

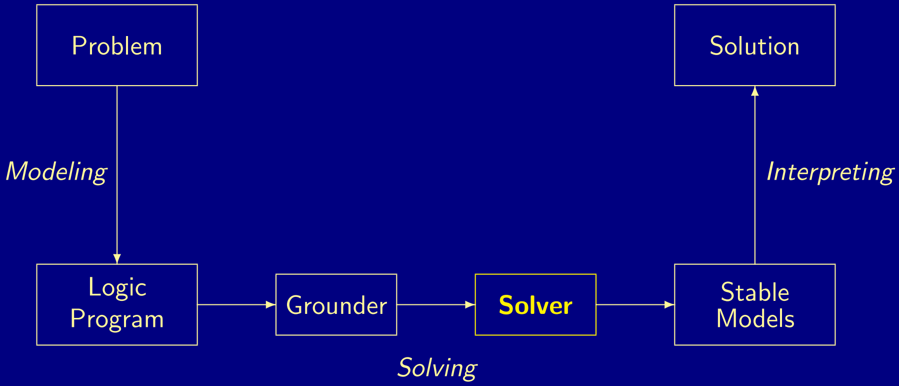
Graph coloring: Grounding

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

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

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

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

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

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

Graph coloring: Grounding

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

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

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

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

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

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

ASP 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)
```

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)
```

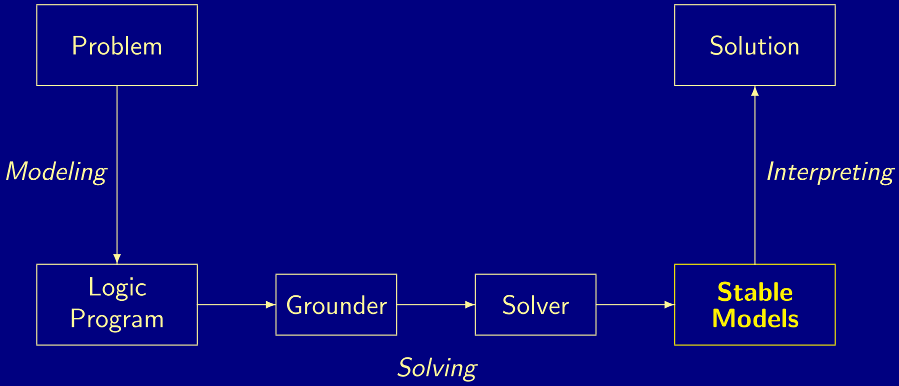
Potassco

Graph coloring: Solving

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

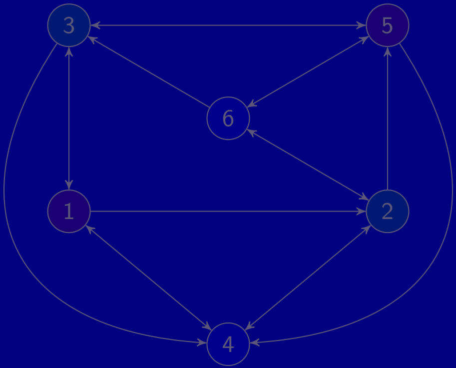
```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
Answer: 3
node(1) [...] assign(6,g) assign(5,b) assign(4,g)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

ASP workflow: Stable models



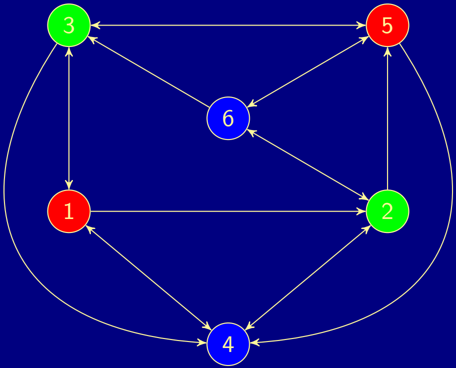
A coloring

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

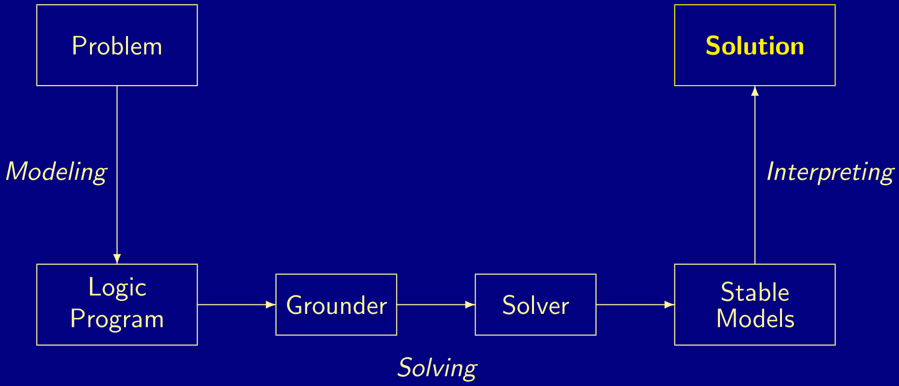


A coloring

```
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

Potassco

Basic methodology

Methodology

Generate and Test (or: Guess and Check)

- **Generator** Generate potential stable model candidates (typically through non-deterministic constructs)
- **Tester** Eliminate invalid candidates (typically through integrity constraints)

Nutshell

Logic program = Data + Generator + Tester (+ Optimizer)

Potassco

Basic methodology

Methodology

Generate and Test (or: Guess and Check)

- **Generator**: Generate potential stable model candidates (typically through non-deterministic constructs)
- **Tester**: Eliminate invalid candidates (typically through integrity constraints)

Nutshell

Logic program $=$ Data $+$ Generator $+$ Tester ($+$ Optimizer)

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Graph coloring

```
node(1..6).

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

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

Data

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

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

Problem encoding

Graph coloring

```
node(1..6).

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

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

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

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

Outline

Outline

Satisfiability testing

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

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

- **Logic Program**

Generator	Tester	Stable models
`{a}.`	`:- 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

Potassco

Satisfiability testing

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

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

- Logic Program

Generator	Tester	Stable models
`{a}.`	`:- 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

Potassco

Satisfiability testing

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

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

- Logic Program

Generator	**Tester**	**Stable models**
`{a}.`	`:- 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

Potassco

Satisfiability testing

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

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

- Logic Program

Generator	**Tester**	**Stable models**
`{a}.`	`:- 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

$$\neg(\neg a \land b) \land \neg(a \land \neg b)$$

- 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

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

- Logic Program

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

Outline

Potassco

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
```

Potassco

Placing some queens

queens.lp

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

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

Placing some queens

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

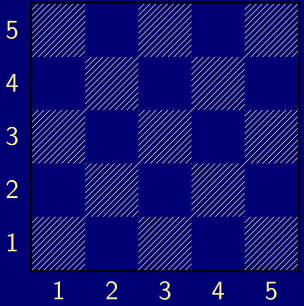
Placing some queens

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

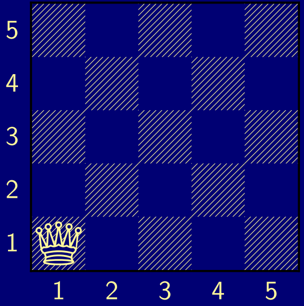
Answer: 1




```
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

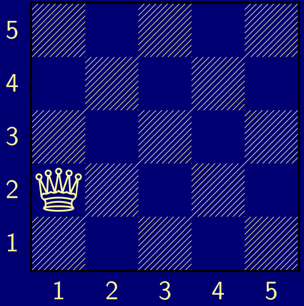
Answer: 2




```
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

Potassco

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

Potassco

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

Potassco

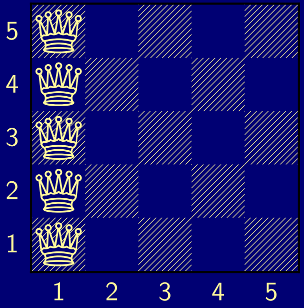
Placing n queens

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

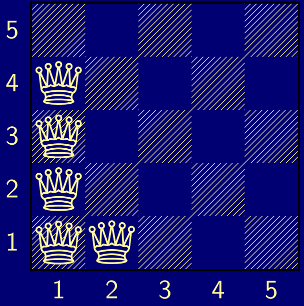
Answer: 1




```
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)
```

Potassco

Horizontal and vertical attack

queens.lp

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

➡ Forbid horizontal and vertical attacks

Potassco

Horizontal and vertical attack

queens.lp

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

➥ Forbid horizontal and vertical attacks

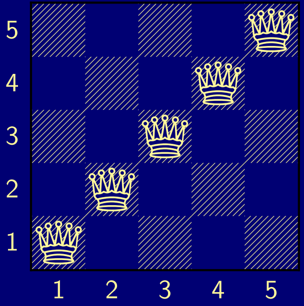
Horizontal and vertical attack

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

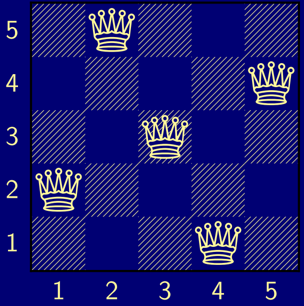
Diagonal attack

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)
```

Potassco

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

Potassco

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

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

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) }.
```

Potassco

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
```

Potassco

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

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) }.
```

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).
[...]

#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) }.
```

Outline

Simplified STRIPS[1] Planning

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

  plan $\langle a, b \rangle$ $\{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

[1] 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 $\langle a, b \rangle$ $\{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

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 $\langle a, b \rangle$ $\{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

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 $\langle a, b \rangle$ $\{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

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).

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

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

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

Simplistic STRIPS Planning

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
[...] occ(a,1) occ(b,2)
SATISFIABLE

Models       : 1
Time         : 0.001s (Solving: 0.00s)
CPU Time     : 0.001s
```

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
[...] occ(a,1) occ(b,2)
SATISFIABLE

Models        : 1
Time          : 0.001s (Solving: 0.00s)
CPU Time      : 0.001s
```

Engineering: Overview

Do it yourself!

- Roland Kaminski, Javier Romero, Torsten Schaub, Philipp Wanko:
  How to build your own ASP-based system?!
  CoRR abs/2008.06692 (2020)

Outline

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
```

Potassco

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
```

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
```

Outline

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)).
```

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)).
```

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
```

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()
```

Embedded script, 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)
```

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
```

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 recent *clingo*-based systems such as *clingcon*, *clingo*[DL], *eclingo*, and *telingo*

Outline

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*

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

clingo = ASP + Control

Extend ASP with dedicated directives

Provide powerful API (here: Python)

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)

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)

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: $(\mathit{ground}^* \mid \mathit{solve}^*)^*$

- Application areas

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

- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

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**)*
- 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

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* $| \ldots)^*$
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

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.`

Potassco

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)
```

Potassco

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)
```

Potassco

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)
```

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

Potassco

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

Potassco

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)
```

An example of incremental solving

```
#program base.
p(0).
#program step (t).
p(t) :- p(t-1).
#program check (t).
#external query(t).        % added in python below
:- not p(42), query(t).
```

An example of incremental solving

```
#program base.
p(0).
#program step (t).
p(t) :- p(t-1).
#program check (t).
#external query(t).        % added in python below
:- not p(42), query(t).
```

An example of incremental solving

```
#program base.
p(0).
#program step (t).
p(t) :- p(t-1).
#program check (t).
#external query(t).        % added in python below
:- not p(42), query(t).
```

An example of incremental solving

```
#program base.
p(0).
#program step (t).
p(t) :- p(t-1).
#program check (t).
#external query(t).        % added in python below
:- not p(42), query(t).
```

Potassco

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_stop(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_stop(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:])
```

Potassco

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]
[...]
```

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
```

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).
```

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).
```

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
```

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

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

Potassco

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

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 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

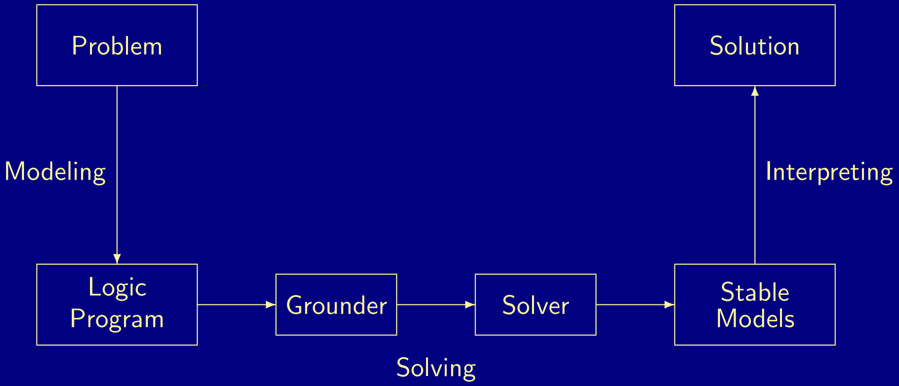
  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Motivation

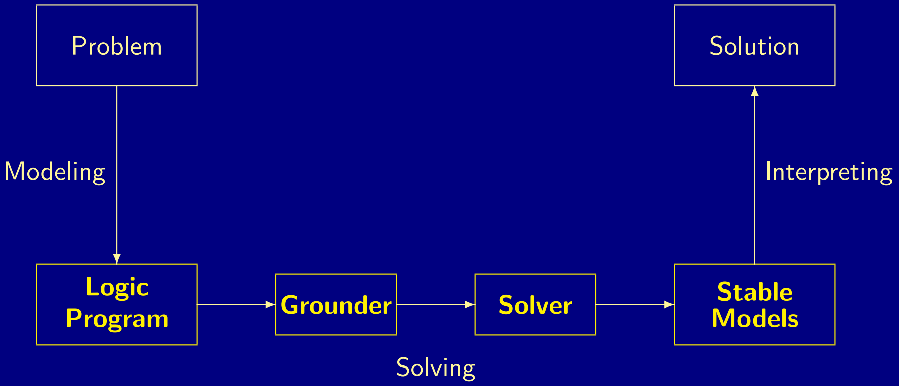
- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories: *ground % theories* | *solve % theories*
- Challenge Logic programs with elusive theory atoms
- Application areas

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

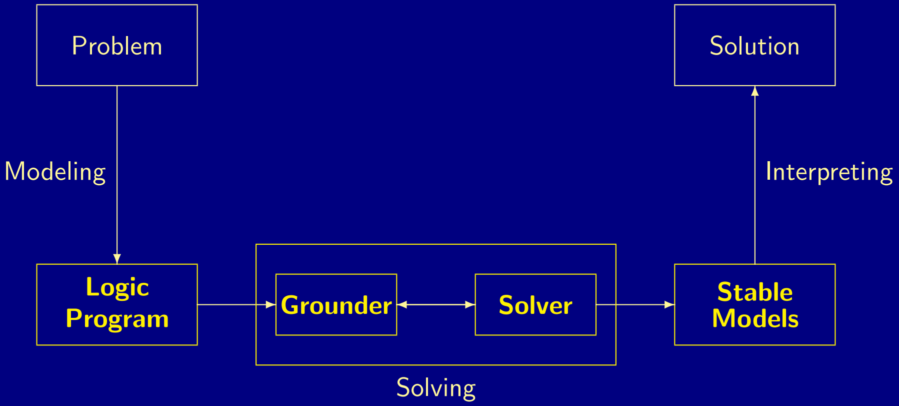
ASP solving process



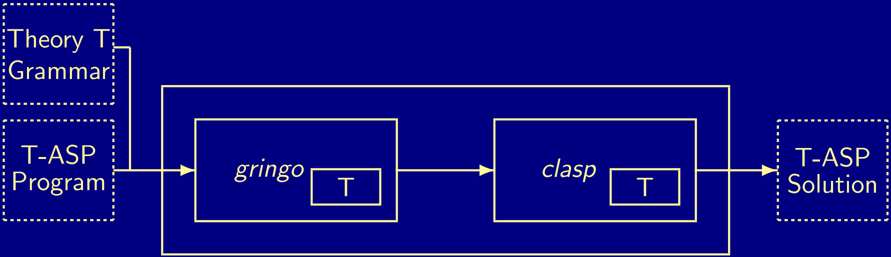
ASP solving process modulo theories



ASP solving process modulo theories

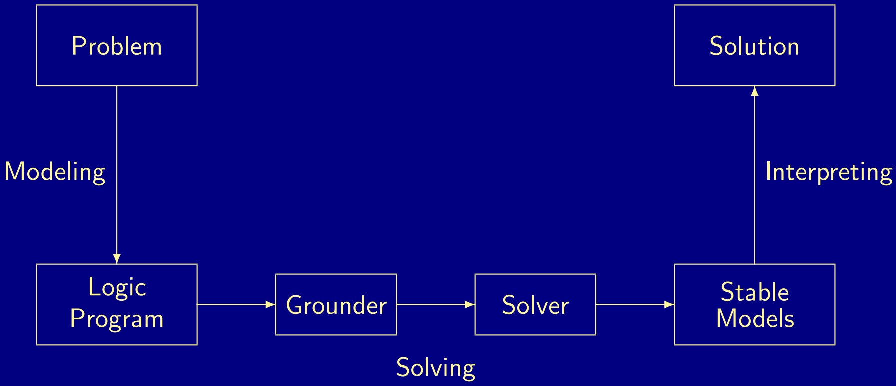


clingo's approach

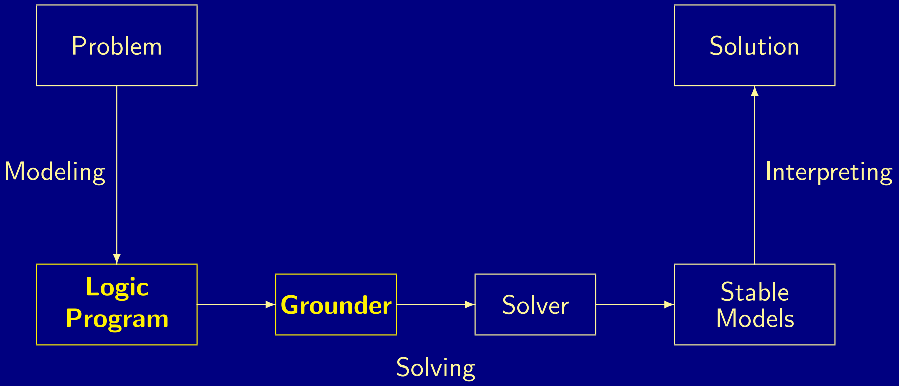


Outline

ASP solving process modulo theories



ASP solving process modulo theories



Linear constraints

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

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

    show_term {
      / : 1, binary, left
    };

    minimize_term {
      + : 5, unary;
      - : 5, unary;
      * : 4, binary, left;
      + : 3, binary, left;
      - : 3, binary, left;
      @ : 0, binary, left
    };
    &dom/0 : dom_term, {=}, linear_term, any;
    &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
    &show/0 : show_term, directive;
    &distinct/0 : linear_term, any;
    &minimize/0 : minimize_term, directive
}.
```

send+more=money

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

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

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

The example has exactly one solution

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

send+more=money

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

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

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

The example has exactly one solution

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

send+more=money

```
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) }.
```

Potassco

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

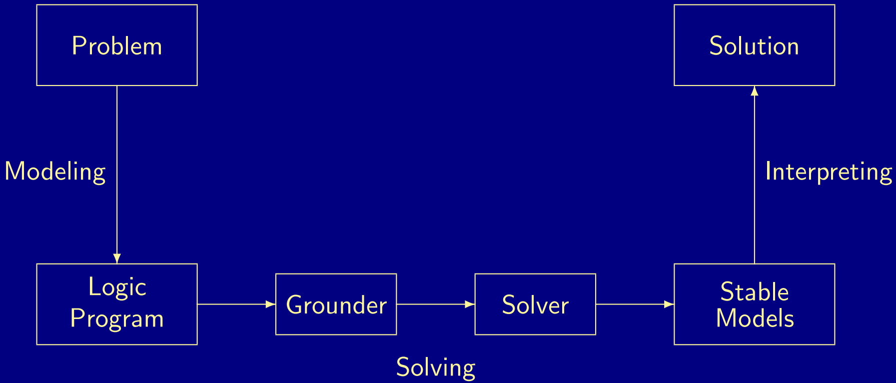
```
digit(1,3,s).   digit(2,3,m).   digit(sum,4,m).
digit(1,2,e).   digit(2,2,o).   digit(sum,3,o).
digit(1,1,n).   digit(2,1,r).   digit(sum,2,n).
digit(1,0,d).   digit(2,0,e).   digit(sum,1,e).
                                digit(sum,0,y).
base(10).
exp(0).  exp(1).  exp(2).  exp(3).  exp(4).
power(1,0).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

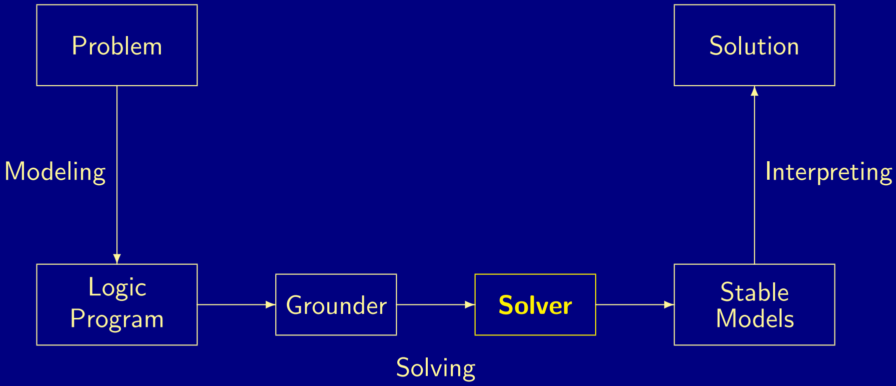
```
UNIX> clingcon sendmoremoney.lp 0
clingcon version 5.0.0
Reading from smm.clp
Solving...
Answer: 1
base(10) exp(0) exp(1) exp(2) exp(3) exp(4)                    \
high(m) high(s) number(1) number(2)                            \
power(1,0) power(10,1) power(100,2) power(1000,3) power(10000,4) \
digit(1,0,d)   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
```

Outline

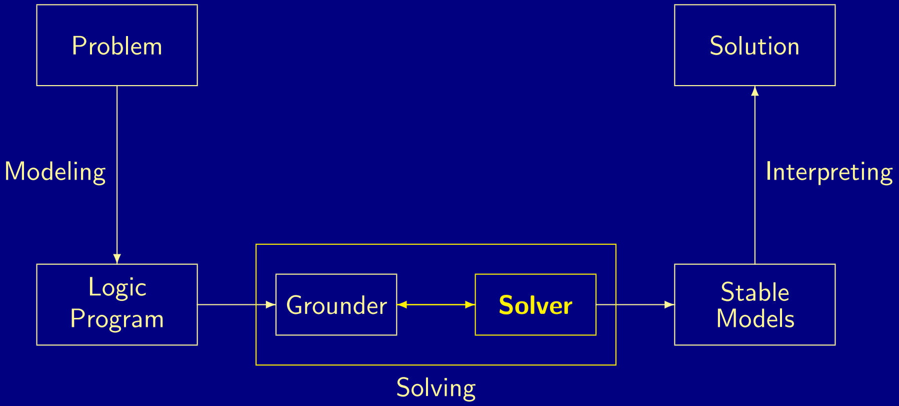
ASP solving process modulo theories



ASP solving process modulo theories



ASP solving process modulo theories

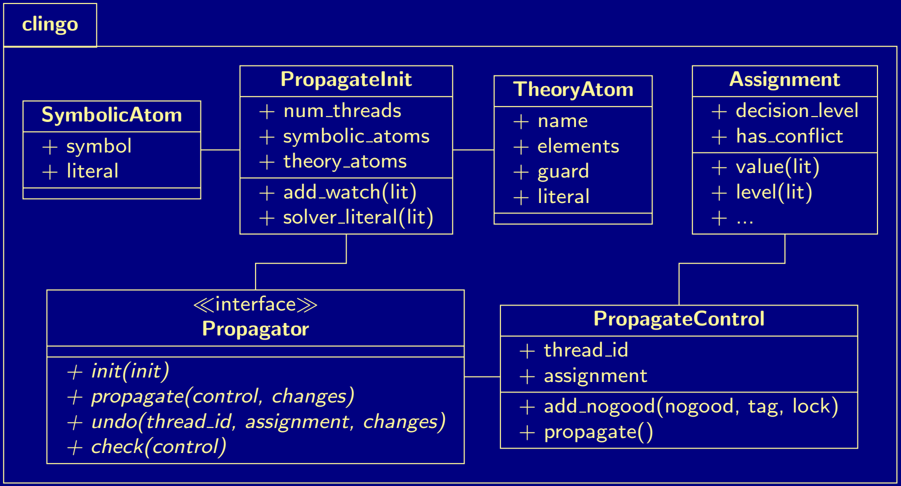


Conflict-driven constraint learning

modulo theories

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

Propagator interface



The *dot* propagator

```
#script (python)

import sys
import time

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

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

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

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

#end.
```

Outline

Motivation

- Observation Sometimes it is advantageous to take a more application-oriented approach by including domain-specific information
  - domain-specific knowledge can be added for improving propagation
  - domain-specific heuristics can be used for making better choices
- Idea Incorporation of domain-specific heuristics by extending
  - input language and/or solver options for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics

Motivation

- Observation Sometimes it is advantageous to take a more application-oriented approach by including domain-specific information
  - domain-specific knowledge can be added for improving propagation
  - domain-specific heuristics can be used for making better choices
- Idea Incorporation of domain-specific heuristics by extending
  - input language and/or solver options for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics

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

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

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \dots, l_n.\ [k@p, m]$

  where

  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier

- Heuristic modifiers

  - `init` for initializing the heuristic value of a with k
  - `factor` for amplifying the heuristic value of a by factor k
  - `level` for ranking all atoms; the rank of a is k
  - `sign` for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Potassco

Heuristic language

- Heuristic directive

  `#heuristic` a : $l_1, \dots, l_n$. $[k@p, m]$

  where
  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  `#heuristic` a : $l_1, \dots, l_n$. $[k@p, m]$

  where
  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a
`true/false`	combine `level` and `sign`

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

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

  `#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]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic occurs(A,T) : action(A), time(T). [2, factor]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic occurs(A,T) : action(A), time(T). [1, level]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic holds(F,T-1) :     holds(F,T). [t-T+1, true]
#heuristic holds(F,T-1) : not holds(F,T)  [t-T+1, false]
                          fluent(F), time(T).
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>   : Modifier
               {1=level|2=pos|3=true|4=neg|
                5=false|6=init|7=factor}
     <pick> : Apply <mod> to
               {0=all|1=scc|2=hcc|4=disj|
                8=min|16=show} atoms
```

Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>   : Modifier
               {1=level|2=pos|3=true|4=neg|
                5=false|6=init|7=factor}
     <pick> : Apply <mod> to
               {0=all|1=scc|2=hcc|4=disj|
                8=min|16=show} atoms
```

- Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>   : Modifier
               {1=level|2=pos|3=true|4=neg|
                5=false|6=init|7=factor}
     <pick> : Apply <mod> to
               {0=all|1=scc|2=hcc|4=disj|
                8=min|16=show} atoms
```

- Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize`$\{a_1, \ldots, a_n\}$
- 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

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

Applications: Overview

Outline

Potassco

Motivation

- Increasing railway traffic demands global and flexible ways for scheduling trains in order to use railway networks to capacity
- Difficulty arises from dependencies among trains induced by connections and shared resources

- Train scheduling combines three distinct tasks
  - Routing
  - Conflict detection and resolution
  - Scheduling

- Solution operational at Swiss Federal Railway using *clingo*[DL]
  - ASP
  - Difference constraints
  - (Hybrid) Optimization
  - Heuristic directives
  - Multi-shot solving

Motivation

- Increasing railway traffic demands global and flexible ways for scheduling trains in order to use railway networks to capacity
- Difficulty arises from dependencies among trains induced by connections and shared resources
- Train scheduling combines three distinct tasks
  - Routing
  - Conflict detection and resolution
  - Scheduling
- Solution operational at Swiss Federal Railway using *clingo*[DL]
  - ASP
  - Difference constraints
  - (Hybrid) Optimization
  - Heuristic directives
  - Multi-shot solving

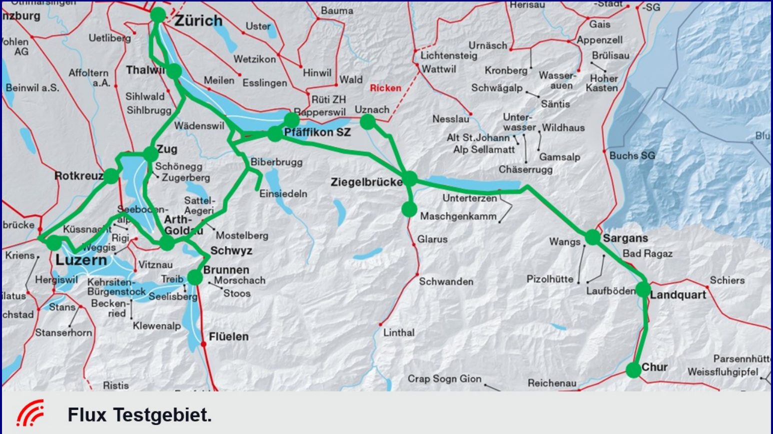
Motivation

- Increasing railway traffic demands global and flexible ways for scheduling trains in order to use railway networks to capacity
- Difficulty arises from dependencies among trains induced by connections and shared resources

- Train scheduling combines three distinct tasks
  - Routing
  - Conflict detection and resolution
  - Scheduling

- Solution operational at Swiss Federal Railway using *clingo*[DL]
  - ASP
  - Difference constraints
  - (Hybrid) Optimization
  - Heuristic directives
  - Multi-shot solving

Benchmark



We optimally solved the train scheduling problem on real-world railway networks spanning about 150 km with up to 467 trains within 5 minutes

Outline

Motivation

- Objective How to develop robust and scalable AI technology for dealing with complex dynamic application scenarios?

- What's needed? — a fruit fly!

  Robotic intra-logistics

- Why?
  - rich multi-faceted, full of variations
  - scalable layout, objects, granularity
  - measurable makespan, energy, quality of service
  - integrative mapf, data, constraints, decisions
  - relevant industry 4.0

- What for? — enabling research and teaching

Potassco

Motivation

- Objective How to develop robust and scalable **KRR** technology for dealing with complex dynamic application scenarios?

- What's needed? — a fruit fly!

  Robotic intra-logistics

- Why?
  - rich multi-faceted, full of variations
  - scalable layout, objects, granularity
  - measurable makespan, energy, quality of service
  - integrative mapf, data, constraints, decisions
  - relevant industry 4.0

- What for? — enabling research and teaching

Motivation

- Objective How to develop robust and scalable KRR technology for dealing with complex dynamic application scenarios?
- What's needed? — a fruit fly!

  Robotic intra-logistics

- Why?
  - rich multi-faceted, full of variations
  - scalable layout, objects, granularity
  - measurable makespan, energy, quality of service
  - integrative mapf, data, constraints, decisions
  - relevant industry 4.0
- What for? — enabling research and teaching

Motivation

- Objective How to develop robust and scalable KRR technology for dealing with complex dynamic application scenarios?
- What's needed? — a model scenario

  Robotic intra-logistics

- Why?
  - rich multi-faceted, full of variations
  - scalable layout, objects, granularity
  - measurable makespan, energy, quality of service
  - integrative mapf, data, constraints, decisions
  - relevant industry 4.0
- What for? — enabling research and teaching

Motivation

- Objective How to develop robust and scalable KRR technology for dealing with complex dynamic application scenarios?
- What's needed? — a model scenario

  Robotic intra-logistics
- Why?
  - rich multi-faceted, full of variations
  - scalable layout, objects, granularity
  - measurable makespan, energy, quality of service
  - integrative mapf, data, constraints, decisions
  - relevant industry 4.0
- What for? — enabling research and teaching

Potassco

Motivation

- Objective How to develop robust and scalable KRR technology for dealing with complex dynamic application scenarios?
- What's needed? — a model scenario

  Robotic intra-logistics
- Why?
  - rich multi-faceted, full of variations
  - scalable layout, objects, granularity
  - measurable makespan, energy, quality of service
  - integrative mapf, data, constraints, decisions
  - relevant industry 4.0
- What for? — enabling research and teaching

Robotic intra-logistics

- Robotics systems for logistics and warehouse automation based on many
  - mobile robots
  - movable shelves
- Main tasks: order fulfillment, i.e.
  - routing
  - order picking
  - replenishment
- Many competing industry solutions:
  - Amazon, Dematic, Genzebach, Gray Orange, Swisslog
- https://youtu.be/TUx-ljgB-5Q

What's (not) in the picture?

- Objects
  floor, robots, shelves, products, people, etc.
- Relations
  positions, carries/d, capacity, orientation, durations, etc.
- Actions
  move, pickup, putdown, pick, charge, restock, etc.
- Objectives
  deadlines, throughput, exploitation, energy management, human machine interaction, etc.

Making robots dance

via temporal and dynamic ASP

- Visit `https://potassco.org/asprilo`

Outline

Potassco

- Potassco Systems — http://potassco.org
  - Academic branch
  - Freely available systems
  - Open source license (MIT)
- Potassco Solutions — http://potassco.com
  - Service branch
  - Consulting
  - Engineering
  - Maintenance
  - Training

Sites Germany (HQ@Potsdam), Australia, Austria, China, Cyprus, Finland, France, Japan, Portugal, Spain, Turkey

Potassco

- Potassco Systems — `http://potassco.org`
  - Academic branch
  - Freely available systems
  - Open source license (MIT)
- Potassco Solutions — `http://potassco.com`
  - Service branch
  - Consulting
  - Engineering
  - Maintenance
  - Training

Sites Germany (HQ@Potsdam), Australia, Austria, China, Cyprus, Finland, France, Japan, Portugal, Spain, Turkey

Outline

The benefits of ASP

It's yours!

- + Transparency
- + Flexibility
- + Maintainability
- + Reliability

- + Generality
- + Effectiveness
- + Optimality
- + Availability

Knowledge

Solver

ASP is a technology,
products emerge from co-operations

Potassco

The benefits of ASP

It's yours!

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Effectiveness
+ Optimality
+ Availability

Knowledge

Solver

ASP is a technology,
products emerge from co-operations

Potassco

The benefits of ASP

It's yours!

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Effectiveness
+ Optimality
+ Availability

Knowledge

Solver

ASP is a technology,
products emerge from co-operations

Potassco

Nutshell

Nutshell

Modeling + Grounding + Solving

Nutshell

Modeling + Grounding + Solving

ASP = DB+LP+KR+SAT

Nutshell

Modeling + Grounding + Solving

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

Nutshell

Modeling + Grounding + Solving

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

`https://potassco.org`

Nutshell

Modeling + Grounding + Solving

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

`https://potassco.org`

And it's fun !

[1] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele.
Potassco User Guide.
University of Potsdam, 2 edition, 2015.

[2] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Answer Set Solving in Practice.
Synthesis Lectures on Artificial Intelligence and Machine Learning.
Morgan and Claypool Publishers, 2012.

[3] R. Kaminski, J. Romero, T. Schaub, and P. Wanko.
How to build your own ASP-based system?!
CoRR, abs/2008.06692, 2020.

[4] V. Lifschitz.
Twelve definitions of a stable model.
In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming*

(ICLP'08), volume 5366 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 2008.

[5] V. Lifschitz.
Thirteen definitions of a stable model.
volume 6300 of *Lecture Notes in Computer Science*, pages 488–503. Springer-Verlag, 2010.

[6] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
ACM Transactions on Computational Logic, 7(2):261–268, 2006.