Answer Set Programming

- Christian Anger
- Kathrin Konczak
- Thomas Linke
- Torsten Schaub

Resources

Course material

- `http://www.cs.uni-potsdam.de/wv/lehre/`
- `http://www.cs.uni-potsdam.de/~torsten/asp/`

Systems

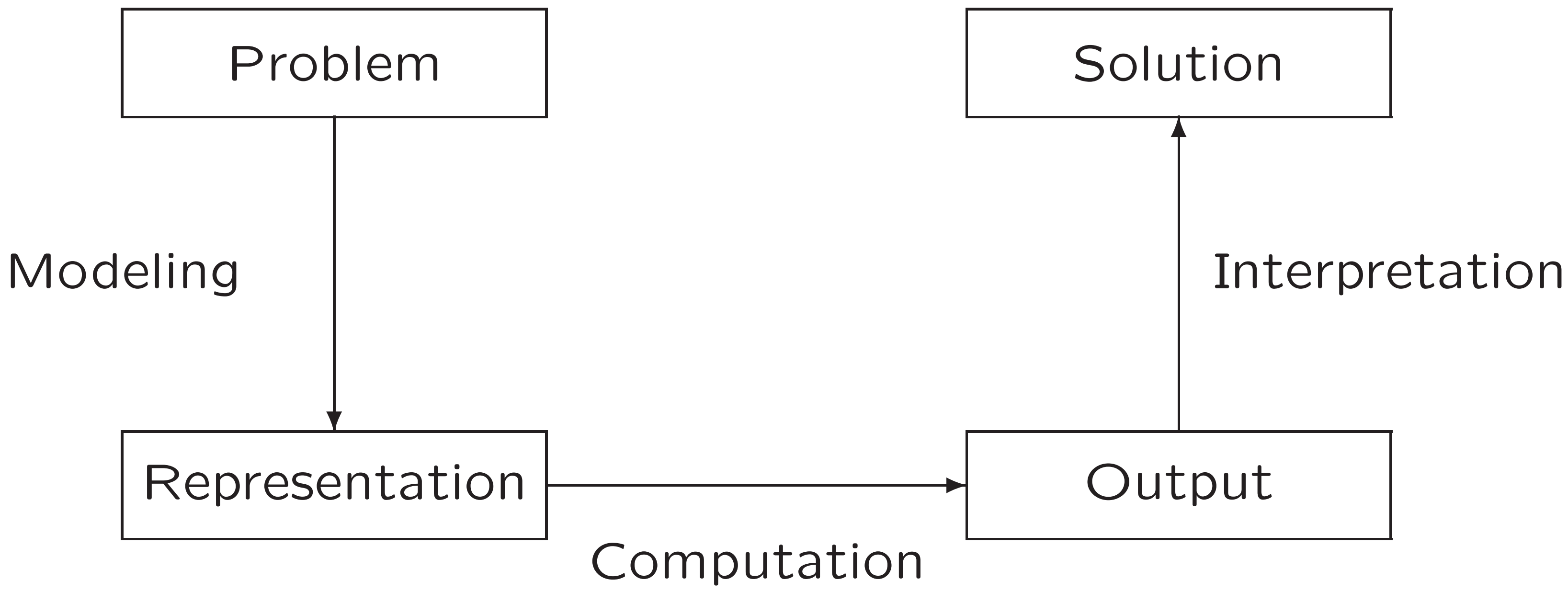
- dlv `http://www.dbai.tuwien.ac.at/proj/dlv/`
- noMoRe `http://www.cs.uni-potsdam.de/~linke/nomore/`
- smodels `http://www.tcs.hut.fi/Software/smodels/`

- plp `http://www.cs.uni-potsdam.de/~torsten/plp/`
- nlp `http://www.cs.uni-potsdam.de/~torsten/nlp/`

Roadmap

- Overview and Introduction
- Modeling
- Extensions: Disjunctive and Nested logic programs
- Extensions: Cardinality and Weight Constraints
- Algorithms and Systems: Smodels
- Algorithms and Systems: noMoRe
- Extensions: Negation and Preferences
- Applications: Configuration
- Applications: Actions and Planning
- Applications: Agents
- Miscellaneous: Dowling/Gallier Algorithm, Fitting's and Well-founded Semantics, Complexity. . .

Problem solving versus Programming



Wanted

An approach to modeling and solving AI problems!

For instance,

- Planning,
- Diagnosis,
- Configuration,
- Combinatorics,
- Puzzles and Games,
- ...

A solution

Answer Set Programming!

Basic Idea

- Encode problem (class+instance) as a set of rules
- Read off solutions from answer sets of the rules

Roots

- Algorithm = Logic + Control (Kowalski, 1979)
- Logic as a programming language
  ↪ Prolog (Colmerauer, Kowalski)
- Related fields
  - Logic Programming (of course)
  - Nonmonotonic Reasoning
  - Deductive Databases
  - Constraint Programming
- Current killer application
  ↪ NASA's space shuttle

Motivation

Prolog (*Pro*gramming in *log*ic) is great, it's *almost* declarative!

To see this, consider

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

and compare it to

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

An interpretation in classical logic amounts to

$$\forall xy(on(x,y) \vee \exists z(on(x,z) \wedge above(z,y)) \rightarrow above(x,y))$$

Motivation (ctd)

Prolog offers *negation as failure* via operator *not*.

For instance,

```
info(a).
ask(X) :- not info(X).
```

cannot be captured by

$$info(a) \land \forall x(\neg info(x) \to ask(x))$$

but by appeal to *Clark's completion* by

$$\begin{aligned} & \forall x(x = a \leftrightarrow info(x)) \land \forall x(\neg info(x) \leftrightarrow ask(x)) \\ \Longleftrightarrow \quad & info(a) \land \forall x(x \neq a \leftrightarrow ask(x)) \end{aligned}$$

Motivation (ctd)

Clark's completion is sometimes too syntactical.

Consider

- `p :- p` yields $p \leftrightarrow p$
- `p :- not p` yields $p \leftrightarrow \neg p$
- Or even more complex yet analogous situations!

Answer Set Programming

- Basic idea
- Syntax
- Semantics
- Examples
- Variables and grounding
- Integrity constraints
- An algorithm

Answer Set Programming: Basic idea

- ✘ View rules as constraints on models.

  For instance, given a rule

  $$c \leftarrow a, \; not \; b$$

  then each model X must satisfy

  if $a \in X$ and $b \notin X$, then $c \in X$.

  ➥ Models are *closed* under rules (informally!).

- ✘ Circumscribe models.

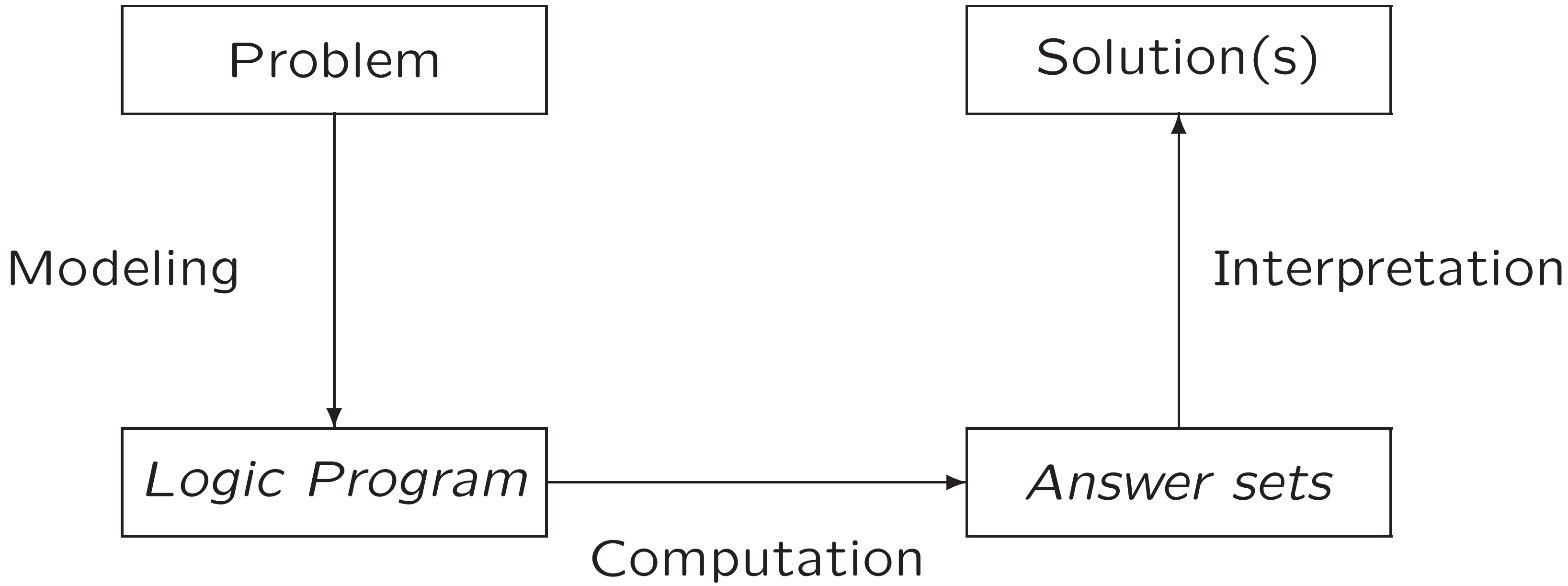
  For instance, prefer model

  $\emptyset$ to $\{a, c\}$ to $\{a, c, d\}$.

  ➥ Consider only *minimal* models containing *justifiable* atoms (informally!).

☞ Such models are called *answer sets* (or initially: *stable models*).

Problem solving in Answer Set Programming



Normal logic programs

- A (normal) *rule*, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n,$$

where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is a atom.

- A (normal) *logic program* is a finite *set* of rules.
- Notation

$$\begin{aligned} head(r) &= A_0 \\ body(r) &= \{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\} \\ body^+(r) &= \{A_1, \ldots, A_m\} \\ body^-(r) &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called *basic* if $body^-(r) = \emptyset$ for all its rules.

Answer sets

Basic programs

- A set of atoms X is *closed under* a basic program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$.
- The *smallest* set of atoms which is closed under a basic program Π is denoted by $Cn(\Pi)$.

Normal programs

- The *reduct*, Π^X, of a program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an *answer set* of a program Π if $Cn(\Pi^X) = X$

A closer look at Π^X

In other words, given a set of atoms X from Π,

Π^X is obtained from Π by deleting

1. each rule having a $not\ A$ in its body with $A \in X$
   and then
2. all negative atoms of the form $not\ A$
   in the bodies of the remaining rules.

Some "logical" remarks

- Basic programs are also referred as *definite clauses*
- Definite clauses are disjunctions with exactly one positive atom
  ➥ $A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$
- *Horn clauses* are clauses with *at most* one positive atom
  ➥ Every definite clause is a Horn clause but not vice versa
- Every set of Horn clauses has a (unique) smallest model.
- This smallest model is the intended semantics of a set of Horn clauses.

☞ Given a basic program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Another "logical" remark

Answer sets versus (minimal) models

- Program $\{a \leftarrow \mathit{not}\ b\}$ has answer set $\{a\}$.
- Clause $a \vee b$ (being equivalent to $a \leftarrow \neg b$)
  - has models $\{a\}$, $\{b\}$, and $\{a, b\}$,
  - among which $\{a\}$ and $\{b\}$ are minimal.

☞ The negation-as-failure operator not makes a difference!

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$$

X	Π	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow p$ $q \leftarrow not\ p$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p\}$	$p \leftarrow p$ $q \leftarrow not\ p$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$ $q \leftarrow not\ p$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p, q\}$	$p \leftarrow p$ $q \leftarrow not\ p$	$p \leftarrow p$	$\emptyset$

A second example

$$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$$

X	Π	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow not\ q$	$p \leftarrow$	$\{p, q\}$
	$q \leftarrow not\ p$	$q \leftarrow$	
$\{p\}$	$p \leftarrow not\ q$	$p \leftarrow$	$\{p\}$
	$q \leftarrow not\ p$		
$\{q\}$	$p \leftarrow not\ q$		$\{q\}$
	$q \leftarrow not\ p$	$q \leftarrow$	
$\{p, q\}$	$p \leftarrow not\ q$		$\emptyset$
	$q \leftarrow not\ p$		

A third example

$\Pi = \{p \leftarrow not\ p\}$

X	Π	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow not\ p$	$p \leftarrow$	$\{p\}$
$\{p\}$	$p \leftarrow not\ p$		$\emptyset$

☞ A program may have zero, one, or multiple answer sets!

A closer look at Cn

Inductive characterization

Let Π be a basic program and X a set of atoms.

- The *immediate consequence operator* T_Π is defined as follows:

$$T_\Pi X = \{head(r) \mid r \in \Pi \text{ and } body(r) \subseteq X\}$$

- Iterated applications of T_Π are written as T_Π^j for $j \geq 0$, where $T_\Pi^0 X = X$ and $T_\Pi^i X = T_\Pi T_\Pi^{i-1} X$ for $i \geq 1$.

Theorem $Cn(\Pi) = \bigcup_{i \geq 0} T_\Pi^i \emptyset$ for any basic program Π.

Proposition $X \subseteq Y$ implies $T_\Pi X \subseteq T_\Pi Y$ for any basic program Π.

↪ In other words, T_Π is monotone.

Let's iterate T_Π

$$\Pi = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

$$\begin{array}{lllllll}
T^0_\Pi \emptyset & = & \emptyset & & & & \\
T^1_\Pi \emptyset & = & \{p, q\} & = & T_\Pi T^0_\Pi \emptyset & = & T_\Pi(\emptyset) \\
T^2_\Pi \emptyset & = & \{p, q, r\} & = & T_\Pi T^1_\Pi \emptyset & = & T_\Pi(\{p, q\}) \\
T^3_\Pi \emptyset & = & \{p, q, r, t\} & = & T_\Pi T^2_\Pi \emptyset & = & T_\Pi(\{p, q, r\}) \\
T^4_\Pi \emptyset & = & \{p, q, r, t, s\} & = & T_\Pi T^3_\Pi \emptyset & = & T_\Pi(\{p, q, r, t\}) \\
T^5_\Pi \emptyset & = & \{p, q, r, t, s\} & = & T_\Pi T^4_\Pi \emptyset & = & T_\Pi(\{p, q, r, t, s\}) \\
T^6_\Pi \emptyset & = & \{p, q, r, t, s\} & = & T_\Pi T^5_\Pi \emptyset & = & T_\Pi(\{p, q, r, t, s\})
\end{array}$$

In fact, $Cn(\Pi) = \{p, q, r, t, s\}$ is the smallest fixpoint of T_Π. That is, $T_\Pi\{p, q, r, t, s\} = \{p, q, r, t, s\}$ and $T_\Pi X \neq X$ for every $X \subseteq \{p, q, r, t, s\}$.

Programs with Variables

Let Π be a logic program.

Herbranduniverse U^Π: Set of constants in Π

Herbrandbase B^Π: Set of (variable-free) atoms constructible from U^Π

☞ We usually denote this as $\mathcal{A}$.

Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U^Π:

$$ground(r) = \{r\theta \mid \theta : var(r) \to U^\Pi\}$$

where $var(r)$ stands for the set of all variables occuring in r;

θ is a (ground) substitution.

Ground Instantiation of Π

$$ground(\Pi) = \{ground(r) \mid r \in \Pi\}$$

An example

$$
\begin{aligned}
\Pi &= \{\ r(a,b) \leftarrow,\ \ r(b,c) \leftarrow,\ \ t(X,Y) \leftarrow r(X,Y)\ \} \\
U^{\Pi} &= \{a,b,c\} \\
B^{\Pi} &= \left\{ \begin{array}{c} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,b), t(b,b), t(b,c), t(c,b), t(c,b), t(c,c) \end{array} \right\}
\end{aligned}
$$

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

Answer sets of programs with Variables

Let Π be a normal logic program with variables.

We define a set X of (ground) atoms as an *answer set* of Π if $Cn(ground(\Pi)^X) = X$.

Programs with Integrity Constraints

Purpose Integrity constraints eliminate unwanted candidate solutions

Syntax An integrity constraints is of the form

$$\leftarrow A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n,$$

where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom.

Example $\leftarrow \text{monitor(21in)}, \text{graphics(evil)}$

Implementation For a new symbol x, map

$$\begin{array}{rrcl} & & \leftarrow & A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n \\ \mapsto & x & \leftarrow & A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n, not\ x \end{array}$$

Another example $\Pi = \{p \leftarrow not\ q,\ q \leftarrow not\ p\}$
versus $\Pi' = \{p \leftarrow not\ q,\ q \leftarrow not\ p,\ \leftarrow p\}$
versus $\Pi'' = \{p \leftarrow not\ q,\ q \leftarrow not\ p,\ \leftarrow not\ p\}$

A first glance at algorithmics

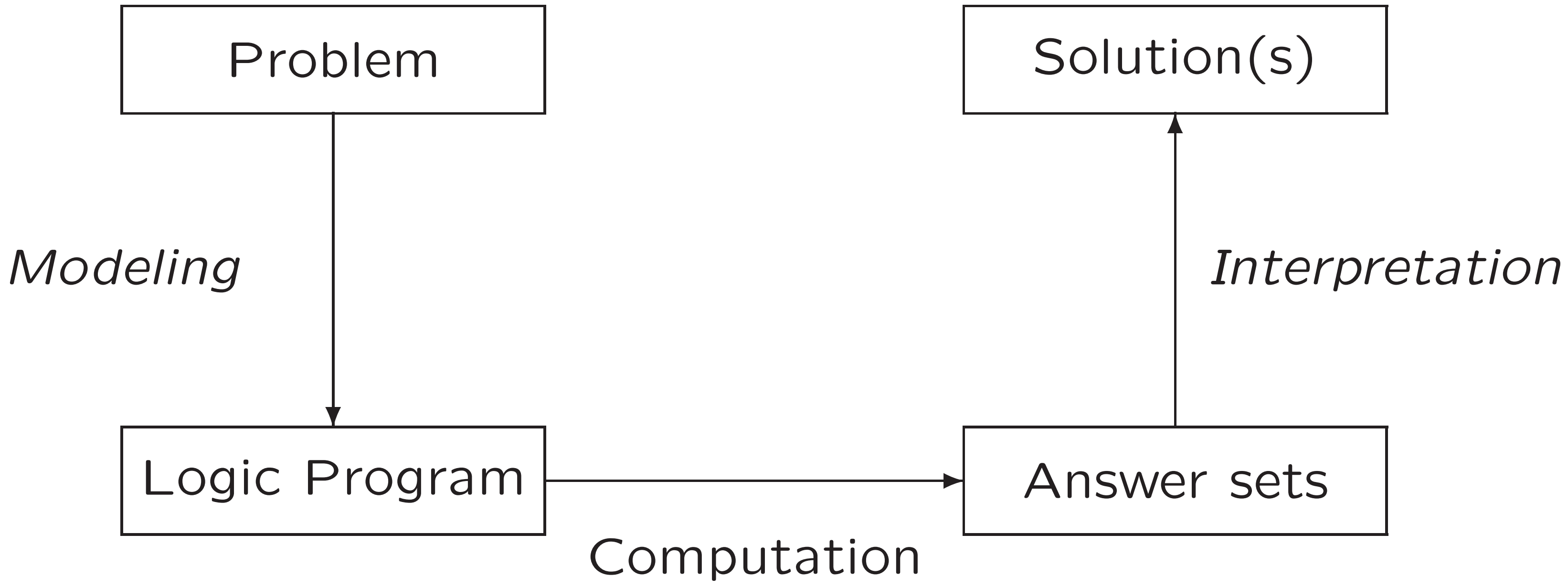
Global parameters: Logic program Π and its set of atoms $\mathcal{A}$:

answerset$_\Pi(X, Y)$

1. $(X, Y) \leftarrow$ *propagation*$_\Pi(X, Y)$
2. if $(X \cap Y) \neq \emptyset$ then fail
3. if $(X \cup Y) = \mathcal{A}$ then return(X)
4. Select $A \in \mathcal{A} \setminus (X \cup Y)$
5. *answerset*$_\Pi(X \cup \{A\}, Y)$
6. *answerset*$_\Pi(X, Y \cup \{A\})$

- (X, Y) is supposed to be a 3-valued model such that $X \subseteq Z$ and $Y \cap Z = \emptyset$ for any answer set Z of Π.
- Key operations: *propagation*$_\Pi(X, Y)$ and 'Select $A \in \mathcal{A} \setminus (X \cup Y)$'
- Worst case complexity: $\mathcal{O}(2^{|\mathcal{A}|})$

Modeling and Interpreting



Problem $\mapsto$ Logic Program

For solving a problem class P for a problem instance I,

encode

1. the problem instance I as a set of facts $C(I)$ and
2. the problem class P as a set of rules $C(P)$,

such that the solutions to P for I can be (polynomially) extracted

from the answer sets of $C(P) \cup C(I)$.

n-colorability of graphs

Problem instance A graph (V, E).

Problem class Assign each vertex in V one of n colors such that no two vertexes in V connected by an edge in E have the same color.

$3-$colorability of graphs

C(I)	vertex(1) $\leftarrow$		edge(1,2) $\leftarrow$	
	vertex(2) $\leftarrow$		edge(2,3) $\leftarrow$	
	vertex(3) $\leftarrow$		edge(3,1) $\leftarrow$	
C(P)	colored(V,r)	$\leftarrow$	not colored(V,b), not colored(V,g), vertex(V)	
	colored(V,b)	$\leftarrow$	not colored(V,r), not colored(V,g), vertex(V)	
	colored(V,g)	$\leftarrow$	not colored(V,r), not colored(V,b), vertex(V),color(C)	
		$\leftarrow$	edge(V,U), colored(V,C),colored(U,C)	
Answer set	{ colored(1,r), colored(2,b), colored(3,g), . . . }			

n-colorability of graphs

with $n = 3$

C(I)	vertex(1) $\leftarrow$	edge(1,2) $\leftarrow$	
	vertex(2) $\leftarrow$	edge(2,3) $\leftarrow$	
	vertex(3) $\leftarrow$	edge(3,1) $\leftarrow$	
C(P)	color(r) $\leftarrow$	color(b) $\leftarrow$	color(g) $\leftarrow$
	colored(V,C)	$\leftarrow$	not othercolor(V,C), vertex(V),color(C)
	othercolor(V,C)	$\leftarrow$	colored(V,C'), C$\neq$C', vertex(V),color(C),color(C')
		$\leftarrow$	edge(V,U),color(C), colored(V,C),colored(U,C)
Answer set	{ colored(1,r), colored(2,b), colored(3,g), ... }		

n–colorability of graphs

with $n = 3$

C(I)
```
vertex(1). vertex(2). vertex(3).
edge(1,2). edge(2,3). edge(3,1).
```

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

colored(V,C)     :- not othercolor(V,C),
                    vertex(V),color(C).
othercolor(V,C) :- colored(V,C1), C != C1,
                    vertex(V),color(C),color(C1).
                 :- edge(V,U),color(C),
                    colored(V,C),colored(U,C).
```

Let it run!

```
torsten@belle-ile 507 > lparse 3color.lp | smodels 0
```

```
smodels version 2.25. Reading...done
Answer: 1
Stable Model: colored(3,g) othercolor(2,g) othercolor(1,g)
othercolor(3,b) colored(2,b) othercolor(1,b) othercolor(3,r)
othercolor(2,r) colored(1,r) color(g) color(b) color(r)
edge(3,1) edge(2,3) edge(1,2) vertex(3) vertex(2) vertex(1)
```

Here's the rest!

```
Answer: 2
Stable Model: colored(3,g) othercolor(2,g) othercolor(1,g) othercolor(3,b)
othercolor(2,b) colored(1,b) othercolor(3,r) colored(2,r) othercolor(1,r)
color(g) color(b) color(r) edge(3,1) edge(2,3) edge(1,2) vertex(3) vertex(2)
vertex(1)
Answer: 3
Stable Model: othercolor(3,g) colored(2,g) othercolor(1,g) colored(3,b)
othercolor(2,b) othercolor(1,b) othercolor(3,r) othercolor(2,r) colored(1,r)
color(g) color(b) color(r) edge(3,1) edge(2,3) edge(1,2) vertex(3) vertex(2)
vertex(1)
Answer: 4
Stable Model: othercolor(3,g) othercolor(2,g) colored(1,g) colored(3,b)
othercolor(2,b) othercolor(1,b) othercolor(3,r) colored(2,r) othercolor(1,r)
color(g) color(b) color(r) edge(3,1) edge(2,3) edge(1,2) vertex(3) vertex(2)
vertex(1)
Answer: 5
Stable Model: othercolor(3,g) colored(2,g) othercolor(1,g) othercolor(3,b)
othercolor(2,b) colored(1,b) colored(3,r) othercolor(2,r) othercolor(1,r)
color(g) color(b) color(r) edge(3,1) edge(2,3) edge(1,2) vertex(3) vertex(2)
vertex(1)
Answer: 6
Stable Model: othercolor(3,g) othercolor(2,g) colored(1,g) othercolor(3,b)
colored(2,b) othercolor(1,b) colored(3,r) othercolor(2,r) othercolor(1,r)
color(g) color(b) color(r) edge(3,1) edge(2,3) edge(1,2) vertex(3) vertex(2)
vertex(1)
False
```

And finally some statistics!

```
Duration: 0.010
Number of choice points: 5
Number of wrong choices: 5
Number of atoms: 28
Number of rules: 45
Number of picked atoms: 42
Number of forced atoms: 0
Number of truth assignments: 347
Size of searchspace (removed): 9 (0)
```

Basic Methodology

Generate and Test (or: Guess and Check) approach

Generator Generate potential candidates answer sets

(typically through non-deterministic constructs)

Tester Eliminate non-valid Candidates

(typically through integrity constraints)

Satisfiability

Problem instance A propositional formula ϕ.

Problem class Is there an assigment of propositional variables to true and false such that a given forumla ϕ is true.

Satisfiability

Consider formula $(a \vee \neg b) \wedge (\neg a \vee b)$.

Generator	**Tester**	**Answer set**
$a \leftarrow not\ \hat{a}$	$\leftarrow not\ a, b$	$A_1 = \{a, b\}$
$\hat{a} \leftarrow not\ a$	$\leftarrow a, not\ b$	$A_2 = \{\hat{a}, \hat{b}\}$
$b \leftarrow not\ \hat{b}$		
$\hat{b} \leftarrow not\ b$		

n-Queens Problem

A solution to $n = 4$:

	Q		
			Q
Q			
		Q	

n-Queens in answer set programming

$q(X, Y)$ gives the legal positions of the queens[a]

$$
\begin{array}{rcl}
q(X,Y) & \leftarrow & not \neg q(X,Y) \\
\neg q(X,Y) & \leftarrow & not\ \ q(X,Y) \\
 & & \\
 & \leftarrow & q(X,Y), q(X',Y), X \neq X' \\
 & \leftarrow & q(X,Y), q(X,Y'), Y \neq Y' \\
 & \leftarrow & q(X,Y), q(X',Y'), |X-X'| = |Y-Y'|, X \neq X', Y \neq Y' \\
 & & \\
 & \leftarrow & not\ \ hasq(X) \\
hasq(X) & \leftarrow & q(X,Y)
\end{array}
$$

[a] regard $\neg q(X, Y)$ as an indepenedant auxiliary atom

n-Queens

(in the `smodels` language)

```
q(X,Y) :- d(X), d(Y), not negq(X,Y).
negq(X,Y) :- d(X), d(Y), not q(X,Y).

:- d(X), d(Y), d(X1), q(X,Y), q(X1,Y),  X1 != X.
:- d(X), d(Y), d(Y1), q(X,Y), q(X,Y1), Y1 != Y.
:- d(X), d(Y), d(X1), d(Y1), q(X,Y), q(X1,Y1),
        X != X1,  Y != Y1, abs(X - X1) == abs(Y - Y1).

:- d(X), not hasq(X).
hasq(X) :-  d(X), d(Y), q(X,Y).

d(1..queens).
```

n-Queens in answer set programming

(in *disjunctive* logic programming)

$$
\begin{array}{rcl}
q(X,Y) \vee \neg q(X,Y) & \leftarrow & \\
& & \\
& \leftarrow & q(X,Y), q(X',Y), X \neq X' \\
& \leftarrow & q(X,Y), q(X,Y'), Y \neq Y' \\
& \leftarrow & q(X,Y), q(X',Y'), |X - X'| = |Y - Y'|, X \neq X', Y \neq Y' \\
& \leftarrow & not\ \ hasq(X) \\
hasq(X) & \leftarrow & q(X,Y)
\end{array}
$$

n-Queens in answer set programming

(in *nested* logic programming)

$$
\begin{array}{rcl}
q(X,Y) \vee not\ q(X,Y) & \leftarrow & \\
& & \\
& \leftarrow & q(X,Y), q(X',Y), X \neq X' \\
& \leftarrow & q(X,Y), q(X,Y'), Y \neq Y' \\
& \leftarrow & q(X,Y), q(X',Y'), |X - X'| = |Y - Y'|, X \neq X', Y \neq Y' \\
& \leftarrow & not\ hasq(X) \\
hasq(X) & \leftarrow & q(X,Y)
\end{array}
$$

n-Queens (ctd)

(in the `smodels` language with cardinality constraints)

$$
\begin{array}{rcl}
1 \; \{ \; q(X,Y) \; \} \; 1 & \leftarrow & d(X) \\
1 \; \{ \; q(X,Y) \; \} \; 1 & \leftarrow & d(Y) \\
\\
& \leftarrow & q(X,Y), q(X',Y'), |X - X'| = |Y - Y'|, X \neq X', Y \neq Y' \\
\\
d(1) & \leftarrow & \\
& \vdots & \\
d(n) & \leftarrow &
\end{array}
$$

n-Queens (ctd)

(in the `smodels` language with cardinality constraints)

```
1 { q(X,Y) : d(Y) } 1 :- d(X).
1 { q(X,Y) : d(X) } 1 :- d(Y).

:- d(X), d(Y), d(X1), d(Y1), q(X,Y), q(X1,Y1), X!=X1,  Y!=Y1,
   abs(X-X1) == abs(Y-Y1).

d(1..queens).
```

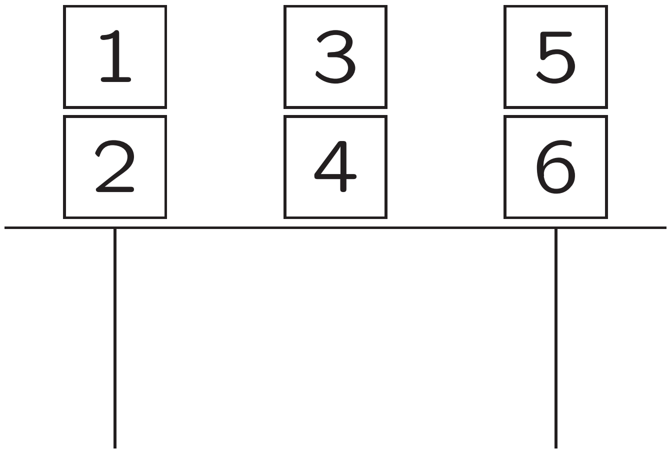
And the Performance ...?

```
torsten@belle-ile 506 > lparse -c queens=20 queens2.lp | smodels
smodels version 2.25. Reading...done
Answer: 1
Stable Model: d(1) d(2) d(3) d(4) d(5) d(6) d(7) d(8) d(9) d(10) d(11) d(12)
d(13) d(14) d(15) d(16) d(17) d(18) d(19) d(20) q(1,16) q(2,13) q(3,6) q(4,3)
q(5,15) q(6,19) q(7,1) q(8,4) q(9,9) q(10,11) q(11,8) q(12,10) q(13,17)
q(14,2) q(15,20) q(16,18) q(17,7) q(18,5) q(19,14) q(20,12)
True
Duration: 37.810
Number of choice points: 1471
Number of wrong choices: 1464
Number of atoms: 501
Number of rules: 10100
Number of picked atoms: 304305
Number of forced atoms: 14604
Number of truth assignments: 3111768
Size of searchspace (removed): 400 (0)
```

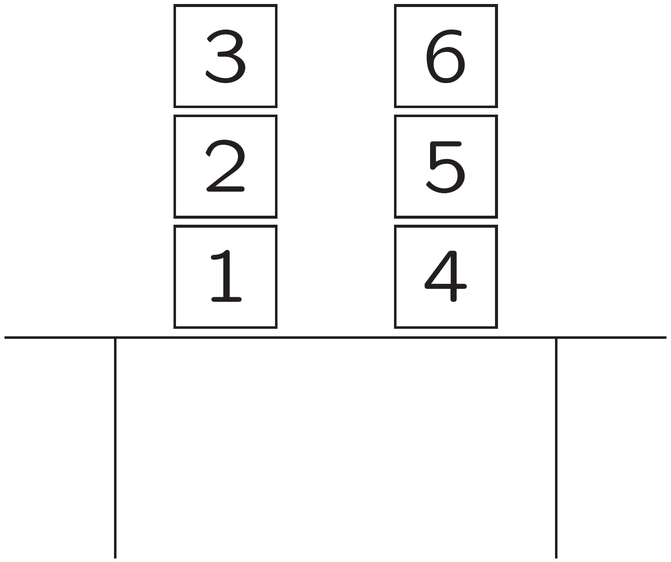
Planning

in the Blocksworld

Initial situation



Goal situation



Initial Situation

```
const grippers=2.
const lasttime=3.

block(1..6).

% DEFINE
on(1,2,0).
on(2,table,0).
on(3,4,0).
on(4,table,0).
on(5,6,0).
on(6,table,0).
```

Goal Situation

```
% TEST
:- not on(3,2,lasttime).
:- not on(2,1,lasttime).
:- not on(1,table,lasttime).
:- not on(6,5,lasttime).
:- not on(5,4,lasttime).
:- not on(4,table,lasttime).
```

Planning in the Blocksworld I

GENERATE

```
time(0..lasttime).

location(B) :- block(B).
location(table).

% GENERATE
{ move(B,L,T) : block(B) : location(L) } grippers :- time(T),
                                                      T<lasttime.
```

Planning in the Blocksworld II

DEFINE

```
% effect of moving a block
on(B,L,T+1) :- move(B,L,T),
               block(B), location(L), time(T), T<lasttime.

% inertia
on(B,L,T+1) :- on(B,L,T), not neg_on(B,L,T+1),
               location(L), block(B), time(T), T<lasttime.

% uniqueness of location
neg_on(B,L1,T) :- on(B,L,T), L!=L1,
                  block(B), location(L), location(L1), time(T).
```

Planning in the Blocksworld III

TEST

```
% neg_on is the negation of on
:- on(B,L,T), neg_on(B,L,T),
   block(B), location(L), time(T).

% two blocks cannot be on top of the same block
:- 2 { on(B1,B,T) : block(B1) },
   block(B), time(T).

% a block can't be moved unless it is clear
:- move(B,L,T), on(B1,B,T),
   block(B), block(B1), location(L), time(T), T<lasttime.

% a block can't be moved onto a block that is being moved also
:- move(B,B1,T), move(B1,L,T),
   block(B), block(B1), location(L), time(T), T<lasttime.
```

The Plan

```
torsten@hoedic 538 > lparse blocks.lp | smodels
smodels version 2.25. Reading...done
Answer: 1
Stable Model: move(1,table,0) move(3,table,0)
              move(2,1,1)     move(5,4,1)
              move(3,2,2)     move(6,5,2)
Duration: 0.050
Number of choice points: 0
Number of wrong choices: 0
Number of atoms: 507
Number of rules: 3026
Number of picked atoms: 24
Number of forced atoms: 13
Number of truth assignments: 944
Size of searchspace (removed): 0 (0)
```

Action description languages

zB. $\mathcal{A}$ (Gelfond & Lifschitz, 1990)

move(b,l) **causes** on(b,l)

inertial on(b,l)

represents

```
% effect of moving a block
on(B,L,T+1) :- move(B,L,T),
               block(B), location(L), time(T), T<lasttime.

% inertia
on(B,L,T+1) :- on(B,L,T), not neg_on(B,L,T+1),
               location(L), block(B), time(T), T<lasttime.
```

Configuration

$$
\begin{array}{rcl}
\text{computer(desktop)} & \leftarrow & \\
\{\ \text{hard-drive(ide)}, \text{hard-drive(scsi)}\ \} & \leftarrow & \text{computer}(x) \\
\text{keyboard(us)} \vee \text{keyboard(german)} & \leftarrow & \text{computer}(x) \\
\text{controller(scsi)} & \leftarrow & \text{hard-drive(scsi)} \\
 & \leftarrow & \text{monitor(21in)} \wedge \text{graphics(evil)}
\end{array}
$$

(Syrjänen, 99) describes a Configurator for Corel Linux!

Disjunctive logic programs

- Syntax
- Semantics
- Examples

Disjunctive logic programs

- A *disjunctive rule*, r, is an ordered pair of the form

$$A_1 \,; \ldots ; A_m \leftarrow A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_o,$$

  where $o \geq n \geq m \geq 0$, and each A_i $(0 \leq i \leq o)$ is a atom.

- A *disjunctive logic program* is a finite set of disjunctive rules.

- (Generalized) Notation

$$\begin{array}{rcl} head(r) & = & \{A_1, \ldots, A_m\} \\ body(r) & = & \{A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_o\} \\ body^+(r) & = & \{A_{m+1}, \ldots, A_n\} \\ body^-(r) & = & \{A_{n+1}, \ldots, A_o\} \end{array}$$

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

Answer sets

Positive programs

- A set of atoms X is *closed under* a positive program Π iff for any $r \in \Pi$, $head(r) \cap X \neq \emptyset$ whenever $body^+(r) \subseteq X$.
- The set of all $\subseteq$-minimal sets of atoms being closed under a positive program Π is denoted by $\min_\subseteq(\Pi)$.

Disjunctive programs

- The *reduct*, Π^X, of a disjunctive program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an answer set of a disjunctive program Π if $X \in \min_\subseteq(\Pi^X)$.

A “positive” example

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

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

3-colorability revisited

C(I)	vertex(1) $\leftarrow$ edge(1,2) $\leftarrow$ vertex(2) $\leftarrow$ edge(2,3) $\leftarrow$ vertex(3) $\leftarrow$ edge(3,1) $\leftarrow$
C(P)	colored(V,r); colored(V,b); colored(V,g) $\leftarrow$ vertex(V) $\leftarrow$ edge(V,U), colored(V,C), colored(U,C)
Answer set	{ colored(1,r), colored(2,b), colored(3,g), . . . }

An example with variables

$$\Pi = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(X)\,;c(Y) & \leftarrow & a(X,Y), not\ c(Y) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), not\ c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), not\ c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), not\ c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), not\ c(2) \end{array} \right\}$$

Clearly, for every answer set X of Π, we have $a(1,2) \in X$ and $\{a(1,1), a(2,1), a(2,2)\} \cap X = \emptyset$.

An example with variables

$$
ground(\Pi)^X \quad = \quad \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\ ; c(1) & \leftarrow & a(1,1), not\ c(1) \\ b(1)\ ; c(2) & \leftarrow & a(1,2), not\ c(2) \\ b(2)\ ; c(1) & \leftarrow & a(2,1), not\ c(1) \\ b(2)\ ; c(2) & \leftarrow & a(2,2), not\ c(2) \end{array} \right\}
$$

- Consider $X = \{a(1,2),\ b(1)\}$.
- We get $\min_{\subseteq}(ground(\Pi)^X) = \{\ \{a(1,2),\ b(1)\},\ \{a(1,2),\ c(2)\}\ \}$.
- X is an answer set of Π because $X \in \min_{\subseteq}(ground(\Pi)^X)$

An example with variables

$$
ground(\Pi)^X \quad = \quad \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), not\ c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), not\ c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), not\ c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), not\ c(2) \end{array} \right\}
$$

- Consider $X = \{a(1,2),\ c(2)\}$.
- We get $\min_\subseteq(ground(\Pi)^X) = \{\ \{a(1,2)\}\ \}$.
- X is no answer set of Π because $X \notin \min_\subseteq(ground(\Pi)^X)$.

Nested logic programs

- Syntax
- Semantics
- Examples

Nested logic programs

- Formulas are formed from
  - propositional atoms and
  - $\top$ and $\bot$

  using
  - negation-as-failure (not),
  - conjunction ($,$), and
  - disjunction ($;$).
- A *nested rule*, r, is an ordered pair of the form $F \leftarrow G$ where F and G are formulas.
- A *nested program* is a finite set of rules.
- Notation: $head(r) = F$ and $body(r) = G$.

Satisfaction relation

- The *satisfaction relation* $X \models F$ between a set of atoms and a formula F is defined recursively as follows:
  - $X \models F$ if $F \in X$ for an atom F
  - $X \models \top$,
  - $X \not\models \bot$,
  - $X \models (F, G)$ if $X \models F$ and $X \models G$,
  - $X \models (F; G)$ if $X \models F$ or $X \models G$,
  - $X \models \mathit{not}\ F$ if $X \not\models F$.

- A set of atoms X satisfies a nested program Π, written $X \models \Pi$, iff for any $r \in \Pi$, $X \models \mathit{head}(r)$ whenever $X \models \mathit{body}(r)$.
- The set of all $\subseteq$-minimal sets of atoms satisfying program Π is denoted by $\min_{\subseteq}(\Pi)$.

Reduct

- The *reduct* F^X of a formula F relative to a set X of atoms is defined recursively as follows:
  - $F^X = F$ if F is an atom or $\top$ or $\bot$,
  - $(F, G)^X = (F^X, G^X)$,
  - $(F; G)^X = (F^X; G^X)$,
  - $(not\ F)^X = \begin{cases} \bot & \text{if } X \models F \\ \top & \text{otherwise} \end{cases}$
- The *reduct*, Π^X, of a nested program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r)^X \leftarrow body(r)^X \mid r \in \Pi\}.$$

- A set X of atoms is an answer set of a nested program Π if $X \in \min_{\subseteq}(\Pi^X)$.

Two examples

- $\Pi_1 = \{(p\,; not\ p) \leftarrow\}$
  - For $X = \emptyset$, we get $\Pi_1^{\emptyset} = \{(p\,; \top) \leftarrow\}$ and $\min_{\subseteq}(\Pi_1^{\emptyset}) = \{\emptyset\}$.
  - For $X = \{p\}$, we get $\Pi_1^{\{p\}} = \{(p\,; \bot) \leftarrow\}$ and $\min_{\subseteq}(\Pi_1^{\{p\}}) = \{\{p\}\}$.
- $\Pi_2 = \{p \leftarrow not\ not\ p\}$
  - For $X = \emptyset$, we get $\Pi_2^{\emptyset} = \{p \leftarrow \bot\}$ and $\min_{\subseteq}(\Pi_2^{\emptyset}) = \{\emptyset\}$.
  - For $X = \{p\}$, we get $\Pi_2^{\{p\}} = \{p \leftarrow \top\}$ and $\min_{\subseteq}(\Pi_2^{\{p\}}) = \{\{p\}\}$.
- In general,
  - $F \leftarrow G,\ not\ not\ H$ is equivalent to $F\,; not\ H \leftarrow G$
  - $F\,;\ not\ not\ G \leftarrow H$ is equivalent to $F \leftarrow\ not\ G, H$
  - $not\ not\ not\ F$ is equivalent to $not\ F$
  - ↪ Intuitionistic Logic HT (Heyting, 1930)

Some more examples

$$
\begin{aligned}
\Pi_3 &= \{p \leftarrow (q, r) \,; (not\ q, not\ s)\} \\
\Pi_4 &= \{(p \,; not\ p), (q \,; not\ q), (r \,; not\ r) \leftarrow\} \\
\Pi_5 &= \{(p \,; not\ p), (q \,; not\ q), (r \,; not\ r) \leftarrow,\ \ \bot \leftarrow p, q\}
\end{aligned}
$$

Implementation

nlp `http://www.cs.uni-potsdam.de/~torsten/nlp`

Language extensions

- Choice rules
- Cardinality constraints
- Weight constraints
- Semantics by embedding in Nested logic programs

Choice rules

Idea Choices over subsets

Syntax

$$\{A_1, \dots, A_m\} \leftarrow A_{m+1}, \dots, A_n, not\ A_{n+1}, \dots, not\ A_o,$$

Informal meaning If the body is satisfied in an answer set, then any subset of $\{A_1, \dots, A_m\}$ can be included in the answer set.

Example The program $\Pi = \{\ \{a\} \leftarrow b,\ b \leftarrow \}$ has two answer sets: $\{b\}$ and $\{a, b\}$.

Implementation smodels

Cardinality constraints

Syntax A cardinality constraint is of the form $l \ \{A_1, \ldots, A_m\} \ u$

Informal meaning A cardinality constraint is satisfied in an answer set X, if the number of atoms from $\{A_1, \ldots, A_m\}$ satisfied in X is between l and u (inclusive).

More formally, if $l \leq |\{A_1, \ldots, A_m\} \cap X| \leq u$.

Conditions $l \ \{A_1, \ldots, A_m : A_{m+1}, \ldots, A_n\} \ u$

where $A_{m+1}, \ldots, A_n$ are often used for type definitions of variables occurring in $A_1, \ldots, A_m$.

Implementation smodels

n-colorability revisited

with $n = 3$

C(I)	vertex(1) $\leftarrow$ edge(1,2) $\leftarrow$ vertex(2) $\leftarrow$ edge(2,3) $\leftarrow$ vertex(3) $\leftarrow$ edge(3,1) $\leftarrow$
C(P)	color(r) $\leftarrow$ color(b) $\leftarrow$ color(g) $\leftarrow$ 1 {colored(V,C):color(C)} 1 $\leftarrow$ vertex(V) $\leftarrow$ edge(V,U),color(C), colored(V,C),colored(U,C)
Answer set	{ colored(1,r), colored(2,b), colored(3,g), ... }

Further extensions

Weight constraints

Syntax $l\ \{A_1 : w_1, \ldots, A_m : w_m\}\ u$

Informal meaning A weight constraint is satisfied in an answer set X, if $l \leq \ \Sigma_{A_i \in X}\, w_i\ \ \leq u$.

Implementation smodels

Optimization

Syntax *minimize* $\{A_1 : w_1, \ldots, A_m : w_m\}$

Implementation smodels

Further extensions (cont'd)

Weak integrity constraints

Syntax $:\sim\ A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n [w : l]$

Informal meaning

1. minimize the sum of weights of violated constraints in the highest level;
2. minimize the sum of weights of violated constraints in the next lower level;
3. etc

Implementation dlv

Embedding in nested logic programs

For formulas $F, F_1, \ldots, F_n$, define

- $\leftarrow F$ as $\bot \leftarrow F$
- $\{F_1 \ldots, F_n\}^c$ as $(F_1 \,; not\ F_1), \ldots, (F_n \,; not\ F_n)$
  - ↪ corresponds to choice over subsets (see above)

For example,

$$\Pi = \left\{ \begin{array}{lll} \{p,q\}^c & \leftarrow & \\ & \leftarrow & p,q \end{array} \right\} \text{ stands for } \left\{ \begin{array}{rll} (p\,; not\ p),(q\,; not\ q) & \leftarrow & \\ \bot & \leftarrow & p,q \end{array} \right\}$$

Embedding in nested logic programs (ctd)

For formulas $F_1, \ldots, F_n$, define

- $l\ \{F_1, \ldots, F_n\}$ as

$$\bigvee_{1 \leq i_1 < \ldots i_l \leq n} (F_{i_1}, \ldots, F_{i_l})$$

- $\{F_1, \ldots, F_n\}\ u$ as $not\ (u + 1\ \{F_1, \ldots, F_n\})$
- $l\ \{F_1, \ldots, F_n\}\ u$ as $(l\ \{F_1, \ldots, F_n\}),\ (\{F_1, \ldots, F_n\}\ u)$

An example

$$\Pi = \left\{ \begin{array}{rcl} \{p,q,r\}^c & \leftarrow & \\ & \leftarrow & 2\ \{p,q,r\} \\ & \leftarrow & \{p,q,r\}\ 0 \end{array} \right\}$$

stands for

$$\left\{ \begin{array}{rcl} (p\,; not\ p) & \leftarrow & \\ (q\,; not\ q) & \leftarrow & \\ (r\,; not\ r) & \leftarrow & \\ \bot & \leftarrow & (p,q)\,;(p,r)\,;(q,r) \\ \bot & \leftarrow & not\ (p\,;q\,;r), \end{array} \right\}$$

Answer sets ? $\{p\}$, $\{q\}$, and $\{r\}$!

Embedding in nested logic programs (ctd)

For formulas $F_1, \ldots, F_n$, define

- $l\ \{F_1, \ldots, F_n\}^c$ as $(\{F_1, \ldots, F_n\}^c, l\ \{F_1, \ldots, F_n\}\ \ )$
- $\ \{F_1, \ldots, F_n\}^c\ u$ as $(\{F_1, \ldots, F_n\}^c,\ \ \{F_1, \ldots, F_n\}\ u)$
- $l\ \{F_1, \ldots, F_n\}^c\ u$ as $(\{F_1, \ldots, F_n\}^c, l\ \{F_1, \ldots, F_n\}\ u)$
  - ↪ corresponds to cardinality constraints (see above)

The program

$$1\ \{p, q, r\}^c\ 1 \leftarrow$$

has the same answer sets as the previous one.

A closer look

$1 \ \{p, q, r\} \ 2 \leftarrow$ versus $1 \ \{p, q, r\}^c \ 2 \leftarrow$

$$
\begin{array}{ll}
\multicolumn{2}{l}{1 \ \{p, q, r\} \ 2 \leftarrow} \\
= & (1 \ \{p, q, r\}), (\{p, q, r\} \ 2) \leftarrow \\
= & (p \,; q \,; r), not \ (3 \ \{p, q, r\}) \leftarrow \\
= & (p \,; q \,; r), not \ (\ p, q, r \ ) \leftarrow
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{1 \ \{p, q, r\}^c \ 2 \leftarrow} \\
= & 1 \ \{p, q, r\} \ 2, \ \ \{p, q, r\}^c \leftarrow \\
= & (p \,; q \,; r), not \ (\ p, q, r \ ), ((p \,; not \ p), (q \,; not \ q), (r \,; not \ r)) \leftarrow
\end{array}
$$

A closer look (ctd)

$$
\begin{aligned}
&(1\ \{p,q,r\}\ 2 \leftarrow)^X \\
&\quad = \quad (1\ \{p,q,r\}\ 2)^X \leftarrow \\
&\quad = \quad (p\ ;q\ ;r)^X, (not\ (\ p,q,r\ ))^X \leftarrow \\
&\quad = \quad (p\ ;q\ ;r), (not\ (\ p,q,r\ ))^X \leftarrow
\end{aligned}
$$

$$
\begin{aligned}
&(1\ \{p,q,r\}^c\ 2 \leftarrow)^X \\
&\quad = \quad (1\ \{p,q,r\}^c\ 2)^X \leftarrow \\
&\quad = \quad (1\ \{p,q,r\}\ 2,\ \{p,q,r\}^c)^X \leftarrow \\
&\quad = \quad (1\ \{p,q,r\}\ 2)^X,\ (\{p,q,r\}^c)^X \leftarrow \\
&\quad = \quad (p\ ;q\ ;r), (not\ (\ p,q,r\ ))^X, ((p\ ;(not\ p)^X), (q\ ;(not\ q)^X), (r\ ;(not\ r)^X)) \leftarrow
\end{aligned}
$$

A closer look (ctd)

Consider $X = \{p, q\}$

$$
\begin{aligned}
& (1\ \{p, q, r\}\ 2 \leftarrow)^{\{p,q\}} \\
& \quad = \quad (p\ ; q\ ; r), (not\ (\ p, q, r\ ))^{\{p,q\}} \leftarrow \\
& \quad = \quad (p\ ; q\ ; r), \top \leftarrow \\
& \quad = \quad (p\ ; q\ ; r) \leftarrow
\end{aligned}
$$

$$
\begin{aligned}
& (1\ \{p, q, r\}^c\ 2 \leftarrow)^{\{p,q\}} \\
& \quad = \quad (p\ ; q\ ; r), (not\ (\ p, q, r\ ))^{\{p,q\}}, ((p\ ; (not\ p)^{\{p,q\}}), \\
& \quad (q\ ; (not\ q)^{\{p,q\}}), (r\ ; (not\ r)^{\{p,q\}})) \leftarrow \\
& \quad = \quad (p\ ; q\ ; r), \top, ((p\ ; \bot), (q\ ; \bot), (r\ ; \top)) \leftarrow \\
& \quad = \quad (p\ ; q\ ; r), (p, q) \leftarrow \\
& \quad = \quad (p, q) \leftarrow
\end{aligned}
$$

What about. . . ?

$s \leftarrow 1\ \{p, q, r\}\ 2$ versus $s \leftarrow 1\ \{p, q, r\}^c\ 2$

Enjoy your exercise $\smile$!

Usage in `lparse` and `smodels`

In `lparse` and `smodels`,

- rule bodies may include
  - $l\ \{F_1, \ldots, F_n\}$
  - $\{F_1, \ldots, F_n\}\ u$
  - $l\ \{F_1, \ldots, F_n\}\ u$
- rule heads may include
  - $\{F_1, \ldots, F_n\}^c$
  - $l\ \{F_1, \ldots, F_n\}^c$
  - $\{F_1, \ldots, F_n\}^c\ u$
  - $l\ \{F_1, \ldots, F_n\}^c\ u$
  - but dropping superscript ^{c}
- For instance, '`{p,q,r} :- {p,q,r} 2`' stands for $\{p, q, r\}^c \leftarrow \{p, q, r\}\ 2$.

Algorithms & Systems: The Smodels approach

- Approximation
- Expansion
- Backtracking search

Approximating answer sets

First Idea Approximate an answer set X by two sets of atoms L and U such that $L \subseteq X \subseteq U$.

- L and U constitute lower and upper bounds on X.
- L and $(\mathcal{A} \setminus U)$ describe a 3-valued model of the program.

Properties Let X be an answer set of normal logic program Π.

- If $L \subseteq X$, then $X \subseteq Cn(\Pi^L)$.
- If $X \subseteq U$, then $Cn(\Pi^U) \subseteq X$.
- If $L \subseteq X \subseteq U$, then $L \cup Cn(\Pi^U) \subseteq X \subseteq U \cap Cn(\Pi^L)$.

Approximating answer sets (ctd)

Second Idea

Iterate

- Replace L by $L \cup Cn(\Pi^U)$
- Replace U by $U \cap Cn(\Pi^L)$

until L and U do not change anymore.

Observations

- At each iteration step
  - L becomes larger (or equal)
  - U becomes smaller (or equal)
- $L \subseteq X \subseteq U$ is invariant for every answer set X of Π
- If $L \not\subseteq U$, then Π has no answer set!
- If $L = U$, then L is an answer set of Π.

The basic expand algorithm

expand(L, U**)**

repeat

$L' \leftarrow L$

$U' \leftarrow U$

$L \leftarrow L' \cup Cn(\Pi^{U'})$

$U \leftarrow U' \cap Cn(\Pi^{L'})$

if $L \not\subseteq U$ **then return**

until $L = L'$ and $U = U'$

☞ Π is a global parameter!

Let's expand!

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, not\ c \\ d \leftarrow b, not\ e \\ e \leftarrow \ not\ d \end{array} \right\}$$

	L'	$Cn(\Pi^{U'})$	L	U'	$Cn(\Pi^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a, b, c, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
2	$\{a\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
3	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$

➥ We have $\{a, b\} \subseteq X$ and $(\mathcal{A} \setminus \{a, b, d, e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every answer set X of Π.

The basic expand algorithm (ctd)

expand

- tightens the approximation on answer sets
- is answer set preserving
- amounts to the well-founded semantics of a program (see following lectures)

Let's expand with d !

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, not\ c \\ d \leftarrow b, not\ e \\ e \leftarrow\ \ not\ d \end{array} \right\}$$

	L'	$Cn(\Pi^{U'})$	L	U'	$Cn(\Pi^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a, d\}$	$\{a, b, c, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$
3	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

➥ $\{a, b, d\}$ is an answer set X of Π.

Let's expand with "$not\ d$" !

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, not\ c \\ d \leftarrow b, not\ e \\ e \leftarrow \ \ not\ d \end{array} \right\}$$

	L'	$Cn(\Pi^{U'})$	L	U'	$Cn(\Pi^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

➥ $\{a, b, e\}$ is an answer set X of Π.

The basic smodels algorithm

smodels(L, U**)**

- **expand(**L, U**)**
- **if** $L \not\subseteq U$ **then return**
- **if** $L = U$ **then exit with** L
- $A \leftarrow$ **select(**$U \setminus L$**)**
- **smodels(**$L \cup \{A\}, U$**)**
- **smodels(**$L, U \setminus \{A\}$**)**

Call: smodels($\emptyset, \mathcal{A}$**)** where $\mathcal{A}$ is the set of all atoms in Π

The basic smodels algorithm (ctd)

- Backtracking search building a binary search tree
- Choice points on atoms
- A node in the search tree corresponds to a 3-valued model
- The search space is pruned by
  - making one choice at a time by appeal to a heuristics (**select**)
  - the set of remaining choices is reduced and conflicts are detected (**expand**)
- Low level implementation using Dowling-Gallier-type data structures (see following lectures)

Outer system architecture

Two-phase implementation

1. `lparse`: Grounding (and handling of "special definitions")
2. `smodels`: Answer set computation for ground programs

Running `smodels`:

```
UNIX> lparse demo.lp | smodels
```

More info:

- `http://www.tcs.hut.fi/Software/smodels/`
- `lparse --help` or `smodels --help`
- Try in particular `lparse -t` and `smodels 0`

What's inside real `smodels`?

expand in `smodels`

- is based on propagation rules
- uses furthermore back-propagation
- generalizes the well-founded semantics
- ...

smodels in `smodels`

- is enhanced by lookahead (see next slide)
- uses smart heuristics
- ...

Lookahead

- Given a program Π, an atom A, and sets of atoms L and U
  - if **expand(**$L \cup \{A\}, U$**)** yields a conflict, then delete A from U
  - if **expand(**$L, U \setminus \{A\}$**)** yields a conflict, then add A to L

- Moreover, lookahead is used for determining the next atom A to *select* (from $U \setminus L$ in the **smodels** algorithm).

  That is, provided that
  - $u^+ = |U \setminus L|$ after **expand(**$L \cup \{A\}, U$**)** and
  - $u^- = |U \setminus L|$ after **expand(**$L, U \setminus \{A\}$**)**

  then

$$h(A) = 2^{u^+} + 2^{u^-}$$

☞ Select an atom A with a minimal value of $h(A)$

Some remarks on `lparse`

- `lparse` accepts only *domain-restricted programs*
- In a program, predicates are (automatically) partitioned into
  - domain predicates
    * no choices
    * no recursion "through" negation-as-failure
  - non-domain predicates
    * all others
- A rule is *domain-restricted* if each of its variables appears in a positive domain predicate in its body.
- A program is *domain-restricted* if every rule is domain-restricted.

Examples for domain predicates

- Facts
  - `vertex(1).  vertex(2).  vertex(3).`
  - `edge(1,2).  edge(2,3).  edge(3,1).`
  - `color(1).  color(2).  color(3).`
- Non-recursive rules
  - `two-edge(X,Y) :- edge(X,Z), edge(Z,Y), not edge(X,Y).`
- Recursive rules
  - `path(X,Y) :- edge(X,Y).`
  - `path(X,Y) :- edge(X,Z), path(Z,Y), vertex(Z).`

Examples for domain-restricted rules

```
colored(V,r)   :-   not colored(V,b), not colored(V,g),
                    vertex(V)
colored(V,b)   :-   not colored(V,r), not colored(V,g),
                    vertex(V)
colored(V,g)   :-   not colored(V,r), not colored(V,b),
                    vertex(V)
               :-   edge(V,U), colored(V,C),colored(U,C),
                    color(C).
```

Domain-restricted programs

- Rules defining domain predicates have a single answer set.

  This answer set can be computed very efficiently through database techniques.

- Domain-restricted programs are grounded in two-steps:

  1. Evaluate domain predicates;
  2. Generate for each rule with variables a set of ground instances that are compatible with the evaluation of domain predicates.

- Also, certain built-ins are supported. For instance,

  – `d(0..n)`

  – `odd(X+1) :- d(X), X<n, not odd(X).`

  where `n` is either explicitly provided in the program by means of `const n=`n or supplied through the command line option `-c n=`n to `lparse`.

Algorithms & Systems: The noMoRe approach

- Exemplary proceeding
- Formal devices
- Coloring sequences

Motivation

Goal Characterize the computation of answer sets of logic programs.

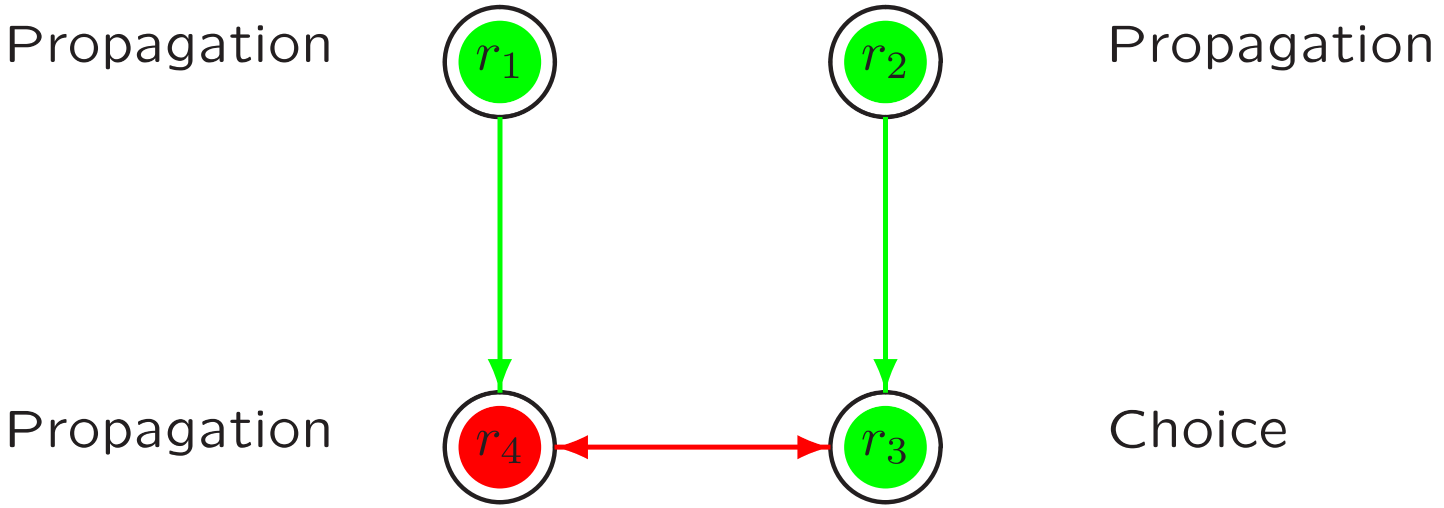
Approach Use rule dependency graphs (RDGs) and their colorings.

Inspiration Proof theory, in particular, SLD derivations.

Outcome A series of operational characterizations of answer sets in terms of operators on partial colorings.

Examplary proceeding

$$
\begin{aligned}
r_1 :&\quad penguin(tweety) &\leftarrow&\\
r_2 :&\quad bird(tweety) &\leftarrow&\\
r_3 :&\quad flies(tweety) &\leftarrow&\ bird(tweety), not\ \neg flies(tweety)\\
r_4 :&\quad \neg flies(tweety) &\leftarrow&\ penguin(tweety), not\ flies(tweety).
\end{aligned}
$$

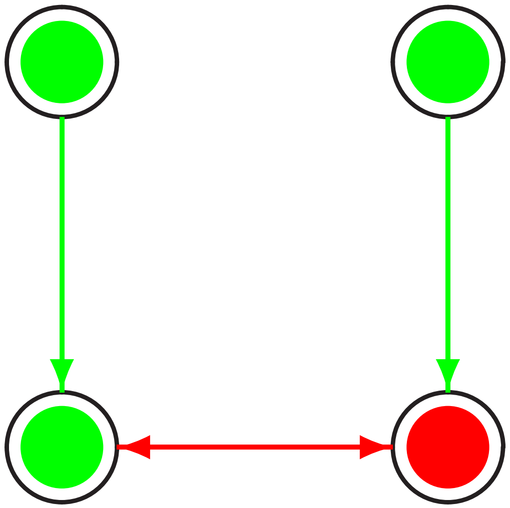


Building graph
Coloring graph
Answer set extraction

$X_1 = \{penguin(tweety), bird(tweety), flies(tweety)\}$

Examplary proceeding

$$
\begin{array}{rrl}
r_1: & penguin(tweety) & \leftarrow \\
r_2: & bird(tweety) & \leftarrow \\
r_3: & flies(tweety) & \leftarrow bird(tweety), not\ \neg flies(tweety) \\
r_4: & \neg flies(tweety) & \leftarrow penguin(tweety), not\ flies(tweety).
\end{array}
$$



$X_2 = \{penguin(tweety), bird(tweety), \neg flies(tweety)\}$

Formal devices in a nutshell

Graphs The rule dependency graph (Π, E_0, E_1) of program Π is a labeled directed graph with

$$E_0 = \{(r, r') \mid r, r' \in \Pi, head(r) \in body^+(r')\} \qquad \text{(0-edges);}$$

$$E_1 = \{(r, r') \mid r, r' \in \Pi, head(r) \in body^-(r')\} \qquad \text{(1-edges);}$$

Colorings are partial functions $C : \Pi \to \{\oplus, \ominus\}$;

➥ **Admissible** colorings are colorings characterizing answer sets;

Operators are *partial* functions $\mathcal{O} : \mathbb{C} \to \mathbb{C}$;

Coloring sequences $(C^i)_{0 \le i \le n}$;

➥ **Successful** coloring sequences enjoy

- C^0 is the "empty" coloring
- $\mathcal{O} : C^i \mapsto C^{i+1}$ for $0 \le i \le n$ and some operator $\mathcal{O}$
- C^n is some admissible coloring

Rule dependency graph

- The *rule dependency graph* (*RDG*)

$$(\Pi, E_0, E_1)$$

of program Π, also written as Γ_Π, is a labeled directed graph with

$$E_0 = \{(r, r') \mid r, r' \in \Pi, head(r) \in body^+(r')\} \qquad \text{(0-edges);}$$

$$E_1 = \{(r, r') \mid r, r' \in \Pi, head(r) \in body^-(r')\} \qquad \text{(1-edges).}$$

An example

- Consider $\Pi = \{r_1, \ldots, r_6\}$, where

$$\begin{array}{rrcl}
r_1 : & p & \leftarrow & \\
r_2 : & b & \leftarrow & p \\
r_3 : & f & \leftarrow & b, not\ f' \\
r_4 : & f' & \leftarrow & p, not\ f \\
r_5 : & b & \leftarrow & m \\
r_6 : & x & \leftarrow & f, f', not\ x
\end{array}$$

- The *RDG* of Π is given as follows:

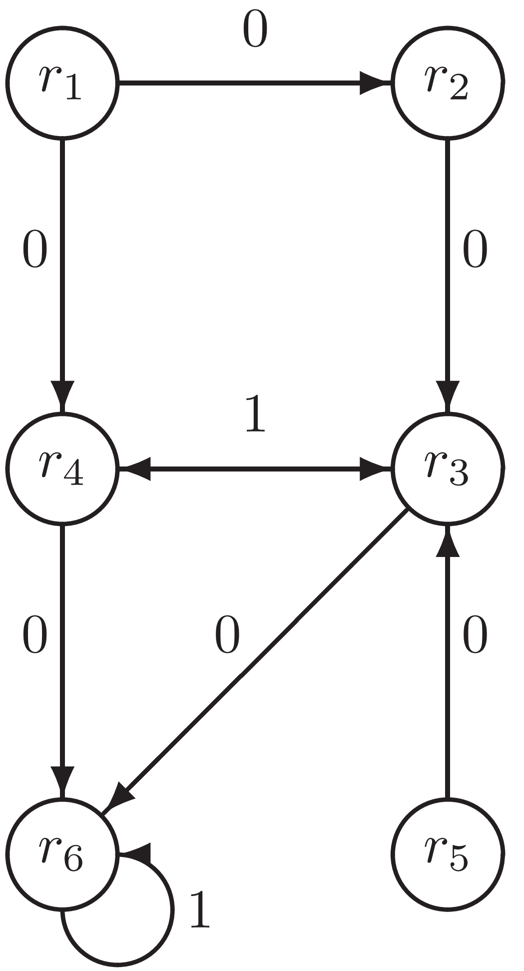
$$\Gamma_\Pi = \left(\Pi, \left\{ \begin{array}{c} (r_1, r_2), (r_1, r_4), (r_2, r_3), \\ (r_3, r_6), (r_4, r_6), (r_5, r_3) \end{array} \right\}, \ \{(r_3, r_4), (r_4, r_3), (r_6, r_6)\} \right)$$

An example (ctd)

- Consider $\Pi = \{r_1, \ldots, r_6\}$, where

$$
\begin{array}{rrcl}
r_1: & p & \leftarrow & \\
r_2: & b & \leftarrow & p \\
r_3: & f & \leftarrow & b, not\ f' \\
r_4: & f' & \leftarrow & p, not\ f \\
r_5: & b & \leftarrow & m \\
r_6: & x & \leftarrow & f, f', not\ x
\end{array}
$$

- The *RDG* of Π is graphically given as:



Coloring

- A *coloring* C of Γ_Π is a mapping $C : \Pi \to \{\oplus, \ominus\}$.
- Define

$$C_\oplus = \{r \mid C(r) = \oplus\} \qquad \text{and} \qquad C_\ominus = \{r \mid C(r) = \ominus\} \ .$$

  We often identify a coloring C with the pair $(C_\oplus, C_\ominus)$.

- If C is *total*, $(C_\oplus, C_\ominus)$ is a binary partition of Π.

  That is, $\Pi = C_\oplus \cup C_\ominus$ and $C_\oplus \cap C_\ominus = \emptyset$.
- A *partial* coloring C induces a pair $(C_\oplus, C_\ominus)$ of sets such that $C_\oplus \cup C_\ominus \subseteq \Pi$ and $C_\oplus \cap C_\ominus = \emptyset$.

Coloring (ctd)

- For comparing partial colorings, C and C', define

$$C \sqsubseteq C' \qquad \text{if} \qquad C_\oplus \subseteq {C_\oplus}' \text{ and } C_\ominus \subseteq {C_\ominus}' \ .$$

- The "empty" coloring $(\emptyset, \emptyset)$ is the $\sqsubseteq$-smallest coloring.
- Accordingly, define

$$C \sqcup C' \qquad \text{as} \qquad (C_\oplus \cup {C_\oplus}', C_\ominus \cup {C_\ominus}') \ .$$

- We denote the set of all partial colorings of a *RDG* Γ_Π by $\mathbb{C}_{\Gamma_\Pi}$.
  ☞ Or simply $\mathbb{C}$ if Γ_Π is clear from the context.

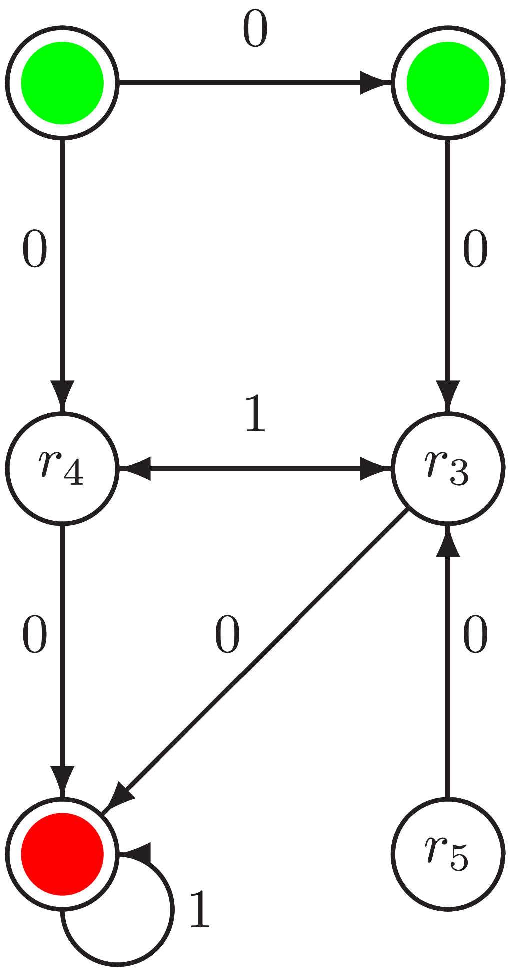
An example

- If C is a coloring of Γ_Π, we call the pair (Γ_Π, C) a *colored* RDG.

- For example, “coloring” the RDG of the previous program Π with

$$C = (\{r_1, r_2\},\ \{r_6\}) = (C_\oplus, C_\ominus)$$

yields the following colored graph.



Colorings representing answer sets

Given a logic program Π along with its *RDG* Γ.

Then, for every answer set X of Π, define a unique *admissible coloring* C of Γ as

$$C = (R_\Pi(X), \Pi \setminus R_\Pi(X))$$

where

$$R_\Pi(X) = \{r \in \Pi \mid \mathit{body}^+(r) \subseteq X \text{ and } \mathit{body}^-(r) \cap X = \emptyset\} \ .$$

Auxiliary concepts

Let $\Gamma = (\Pi, E_0, E_1)$ be the *RDG* of logic program Π and C be a partial coloring of Γ.

For $r \in \Pi$, define:

1. r is *supported* in (Γ, C), if $body^+(r) \subseteq \{head(r') \mid (r', r) \in E_0, r' \in C_\oplus\}$;
2. r is *unsupported* in (Γ, C), if $\{r' \mid (r', r) \in E_0, head(r') = q\} \subseteq C_\ominus$ for some $q \in body^+(r)$;
3. r is *blocked* in (Γ, C), if $r' \in C_\oplus$ for some $(r', r) \in E_1$;
4. r is *unblocked* in (Γ, C), if $r' \in C_\ominus$ for all $(r', r) \in E_1$.

Whenever C is total, a rule is unsupported (or unblocked) iff it is not supported (or not blocked, respectively).

Auxiliary concepts (ctd)

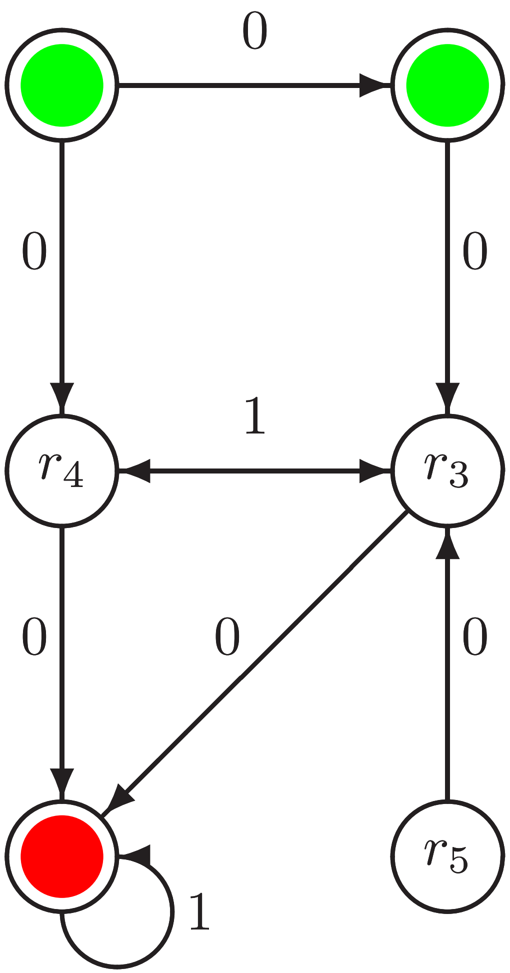
Let Γ be the *RDG* of program Π and C be a partial coloring of Γ.

We define

1. $S(\Gamma, C) = \{r \in \Pi \mid r$ is supported in $(\Gamma, C)\}$;
2. $\overline{S}(\Gamma, C) = \{r \in \Pi \mid r$ is unsupported in $(\Gamma, C)\}$;
3. $B(\Gamma, C) = \{r \in \Pi \mid r$ is blocked in $(\Gamma, C)\}$;
4. $\overline{B}(\Gamma, C) = \{r \in \Pi \mid r$ is unblocked in $(\Gamma, C)\}$.

An example (ctd)

- Consider the sets obtained regarding the above colored *RDG*.
- We get:
  - $S(\Gamma_\Pi, C) = \{r_1, r_2, r_3, r_4\}$
  - $\overline{S}(\Gamma_\Pi, C) = \{r_5\}$
  - $B(\Gamma_\Pi, C) = \emptyset$
  - $\overline{B}(\Gamma_\Pi, C) = \{r_1, r_2, r_5, r_6\}$



☞ Rules like $r_1 = p \leftarrow$ or $r_5 = b \leftarrow m$ must be distinguished through their "inner" structure!

Relation to answer sets

Let C be the admissible coloring of Γ corresponding to answer set X of Π.

For $r \in \Pi$, we have

1. $r \in S(\Gamma, C)$ iff $body^+(r) \subseteq X$;
2. $r \in \overline{S}(\Gamma, C)$ iff $body^+(r) \not\subseteq X$;
3. $r \in B(\Gamma, C)$ iff $body^-(r) \cap X \neq \emptyset$;
4. $r \in \overline{B}(\Gamma, C)$ iff $body^-(r) \cap X = \emptyset$.

Relation to answer sets (ctd)

Let Γ be the *RDG* of logic program Π and C be a partial coloring of Γ.

If C is admissible, we have for the corresponding answer set X of Π that

1. $S(\Gamma, C) \cap \overline{B}(\Gamma, C) = R_\Pi(X)$;
2. $\overline{S}(\Gamma, C) \cup B(\Gamma, C) = \Pi \setminus R_\Pi(X)$.

For every answer set X of Π with

$$C_\oplus \subseteq R_\Pi(X) \qquad \text{and} \qquad C_\ominus \subseteq \Pi \setminus R_\Pi(X)$$

we have that

1. $S(\Gamma, C) \cap \overline{B}(\Gamma, C) \subseteq R_\Pi(X)$;
2. $\overline{S}(\Gamma, C) \cup B(\Gamma, C) \subseteq \Pi \setminus R_\Pi(X)$.

Propagation operator $\mathcal{P}$

Let Γ be the *RDG* of logic program Π and C be a partial coloring of Γ.

- Define

  $$\mathcal{P}_\Gamma : \mathbb{C} \rightarrow \mathbb{C}$$

  as

  $$\mathcal{P}_\Gamma(C) = C \sqcup (S(\Gamma, C) \cap \overline{B}(\Gamma, C),\ \overline{S}(\Gamma, C) \cup B(\Gamma, C)) \ .$$

- A partial coloring C is closed under $\mathcal{P}_\Gamma$, if $C = \mathcal{P}_\Gamma(C)$.

Propagation operator $\mathcal{P}$ (ctd)

- Define

  $$\mathcal{P}^*_\Gamma : \mathbb{C} \to \mathbb{C}$$

  where $\mathcal{P}^*_\Gamma(C)$ is

  - the $\sqsubseteq$-smallest partial coloring
  - containing C and
  - being closed under $\mathcal{P}_\Gamma$.

- Alternatively, we have

  $$\mathcal{P}^*_\Gamma(C) = \bigsqcup_{i<\omega} P^i(C)$$

  where

  1. $P^0(C) = C$ and
  2. $P^{i+1}(C) = \mathcal{P}_\Gamma(P^i(C))$ for $i < \omega$.

An example

$$\begin{array}{llllllllllll} r_1: & p & \leftarrow & & r_3: & f & \leftarrow & b, not\ f' & r_5: & b & \leftarrow & m \\ r_2: & b & \leftarrow & p & r_4: & f' & \leftarrow & p, not\ f & r_6: & x & \leftarrow & f, f', not\ x \end{array}$$

$$\begin{array}{rcl} \mathcal{P}_\Gamma((\emptyset,\emptyset)) & = & (\emptyset,\emptyset) \sqcup (\{r_1\} \cap \{r_1, r_2, r_5\}, \{r_5\} \cup \emptyset) \\ & = & (\{r_1\}, \{r_5\}) \\ \mathcal{P}_\Gamma((\{r_1\}, \{r_5\})) & = & (\{r_1\}, \{r_5\}) \sqcup (\{r_1, r_2, r_4\} \cap \{r_1, r_2, r_5\}, \{r_5\} \cup \emptyset) \\ & = & (\{r_1, r_2\}, \{r_5\}) \\ \mathcal{P}_\Gamma((\{r_1, r_2\}, \{r_5\})) & = & (\{r_1, r_2\}, \{r_5\}) \sqcup (\{r_1, r_2, r_3, r_4\} \cap \{r_1, r_2, r_5\}, \{r_5\} \cup \emptyset) \\ & = & (\{r_1, r_2\}, \{r_5\}) \end{array}$$

Hence, we obtain $\mathcal{P}^*_\Gamma((\emptyset,\emptyset)) = (\{r_1, r_2\}, \{r_5\})$.

Propagation operator $\mathcal{U}$

- Originally, $\mathcal{U}$ is defined in graph-theoretical terms by means of the so-called "support graph".[a]

- Alternative definition:

  Define $\mathcal{U}_\Gamma : \mathbb{C} \rightarrow \mathbb{C}$ as

  $$\mathcal{U}_\Gamma(C) = (C_\oplus, C_\ominus \cup \{r \mid \mathit{body}^+(r) \not\subseteq \mathit{Cn}((\Pi \setminus C_\ominus)^\emptyset)\}) \ .$$

- For instance, for $\Pi = \{p \leftarrow q, q \leftarrow p\}$, we obtain

  $$\mathcal{U}_\Gamma((\emptyset,\emptyset)) = (\emptyset, \{p \leftarrow q, q \leftarrow p\}) \ ,$$

  which is not obtainable through $\mathcal{P}_\Gamma$.

[a]Intuitively, support graphs constitute the graph-theoretical counterpart of *Cn*.

Choice operators

Let Γ be the *RDG* of logic program Π and C be a partial coloring of Γ.

- For $\circ \in \{\oplus, \ominus\}$, define $\mathcal{C}_\Gamma^\circ : \mathbb{C} \to \mathbb{C}$ as
  1. $\mathcal{C}_\Gamma^\oplus(C) = (C_\oplus \cup \{r\}, C_\ominus)$ for some $r \in \Pi \setminus (C_\oplus \cup C_\ominus)$;
  2. $\mathcal{C}_\Gamma^\ominus(C) = (C_\oplus, C_\ominus \cup \{r\})$ for some $r \in \Pi \setminus (C_\oplus \cup C_\ominus)$.

- For $\circ \in \{\oplus, \ominus\}$, define $\mathcal{D}_\Gamma^\circ : \mathbb{C} \to \mathbb{C}$ as
  1. $\mathcal{D}_\Gamma^\oplus(C) = (C_\oplus \cup \{r\}, C_\ominus)$ for some $r \in S(\Gamma, C) \setminus (C_\oplus \cup C_\ominus)$;
  2. $\mathcal{D}_\Gamma^\ominus(C) = (C_\oplus, C_\ominus \cup \{r\})$ for some $r \in S(\Gamma, C) \setminus (C_\oplus \cup C_\ominus)$.

All operators at a glance

- $\mathcal{P}$ and $\mathcal{P}^*$:
  - deterministic
  - provide basic forward propagation
  - $\mathcal{P}^*$ computes the closure under $\mathcal{P}$
  - $\mathcal{P}^*$ amounts to closure under Fitting's operator (see below)
  - $\mathcal{P}$ is reflexive, monotonic, and answer set preserving
- $\mathcal{U}$ and $\mathcal{V}$:
  - deterministic
  - allow for detecting unsupported rules
  - $\mathcal{V}$ is an incremental variant of $\mathcal{U}$
  - $(\mathcal{PU})^*$ amounts to well-founded semantics (see below)
  - $\mathcal{U}$ is reflexive, idempotent, monotonic, and answer set preserving

All operators at a glance (ctd)

- $\mathcal{N}$:
  - deterministic
  - colors all uncolored rules with $\ominus$
  - Formally: $\mathcal{N}_\Gamma(C) = (C_\oplus, \Pi \setminus C_\oplus)$.
- $\mathcal{C}^\circ$ and $\mathcal{D}^\circ$ for $\circ \in \{\oplus, \ominus\}$:
  - nondeterministic
  - choose a rule and color it, either with $\oplus$ or $\ominus$
  - $\mathcal{D}^\circ$ is restricted to choosing supported rules only

Operational answer set characterization I

Let Γ be the *RDG* of logic program Π and let C be a total coloring of Γ.

Then, C is an admissible coloring of Γ

iff

there exists a sequence $(C^i)_{0\le i\le n}$ with the following properties:

1. $C^0 = (\emptyset, \emptyset)$;
2. $C^{i+1} = \mathcal{C}^{\circ}_{\Gamma}(C^i)$ for some $\circ \in \{\oplus, \ominus\}$ and $0 \le i < n$;
3. $C^n = \mathcal{P}_{\Gamma}(C^n)$;
4. $C^n = \mathcal{U}_{\Gamma}(C^n)$;
5. $C^n = C$.

Operational answer set characterization II

Let Γ be the *RDG* of logic program Π and let C be a total coloring of Γ.

Then, C is an admissible coloring of Γ

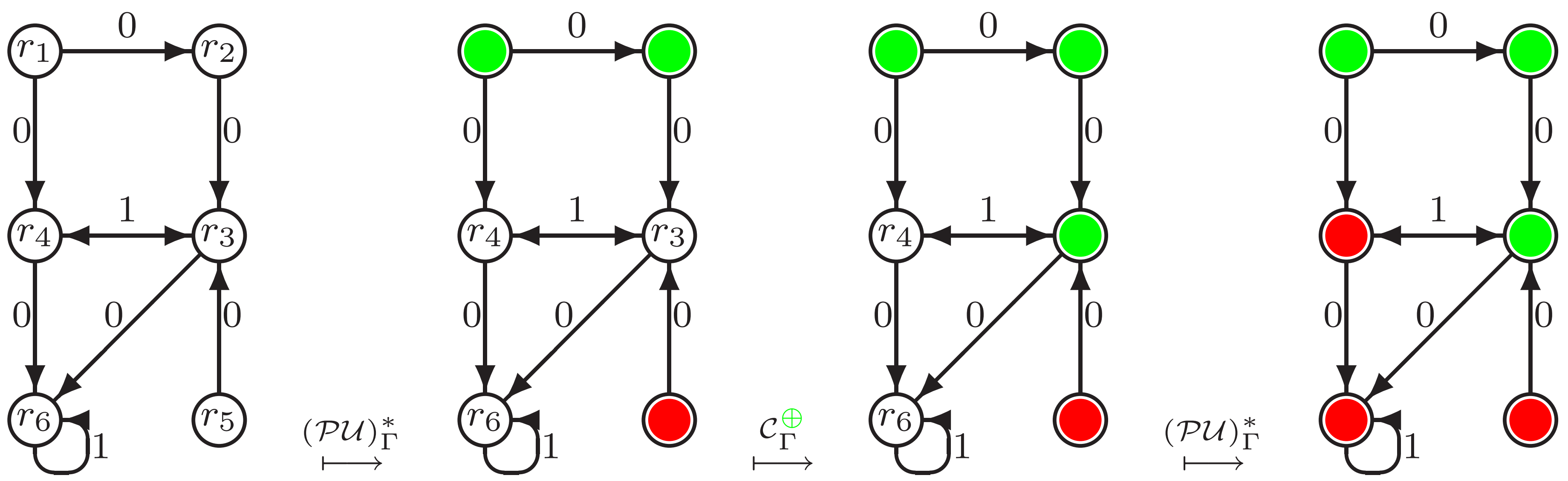
iff

there exists a sequence $(C^i)_{0\leq i\leq n}$ with the following properties:

1. $C^0 = (\mathcal{PU})^*_\Gamma((\emptyset,\emptyset))$;
2. $C^{i+1} = (\mathcal{PU})^*_\Gamma(\mathcal{C}^\circ_\Gamma(C^i))$ for some $\circ \in \{\oplus,\ominus\}$ and $0\leq i<n$;
3. $C^n = C$.

A coloring sequence according to II

$$
\begin{array}{llll@{\qquad}llll@{\qquad}llll}
r_1: & p & \leftarrow & & r_3: & f & \leftarrow & b, not\ f' & r_5: & b & \leftarrow & m \\
r_2: & b & \leftarrow & p & r_4: & f' & \leftarrow & p, not\ f & r_6: & x & \leftarrow & f, f', not\ x
\end{array}
$$



Operational answer set characterization V

Let Γ be the *RDG* of logic program Π and let C be a total coloring of Γ.

Then, C is an admissible coloring of Γ

iff

there exists a sequence $(C^i)_{0\leq i\leq n}$ with the following properties:

1. $C^0 = \mathcal{P}^*_\Gamma((\emptyset,\emptyset))$;
2. $C^{i+1} = \mathcal{P}^*_\Gamma(\mathcal{D}^\circ_\Gamma(C^i))$ where $\circ \in \{\oplus,\ominus\}$ and $0 \leq i < n-1$;
3. $C^n = \mathcal{N}_\Gamma(C^{n-1})$;
4. $C^n = \mathcal{P}_\Gamma(C^n)$;
5. $C^n = C$.

☞ This is the basic strategy of the `noMoRe` system!

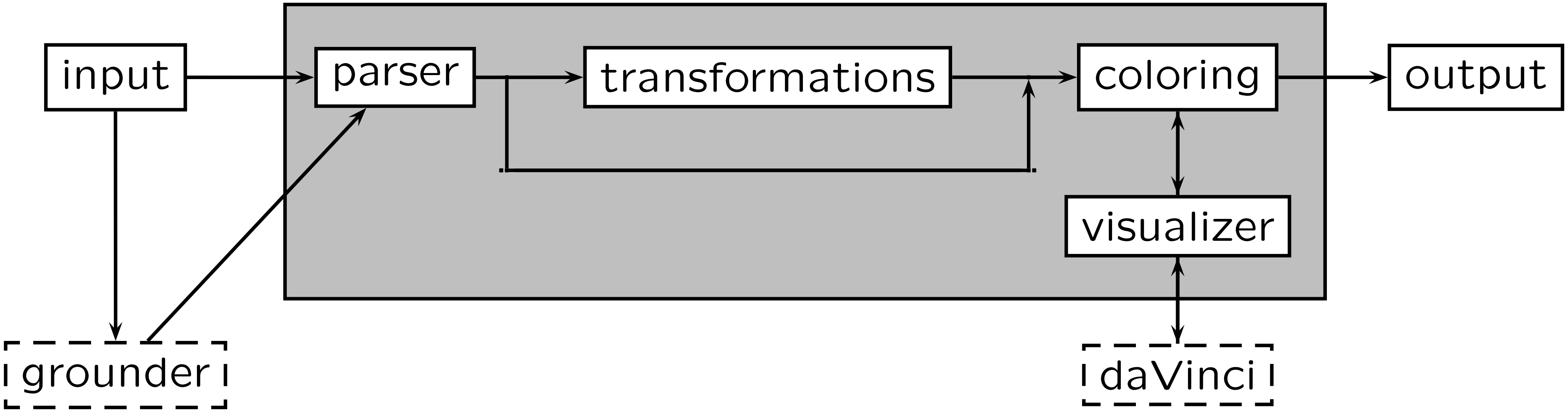
Summary of operational characterizations

	Formation Process	Check	Properties	Properties Prefix sequences
I	$[\mathcal{C}_\Gamma^\circ]^k$	$\mathcal{P}_\Gamma, \mathcal{U}_\Gamma$	1–5	1–3
II	$[(\mathcal{PU})_\Gamma^* \circ \mathcal{C}_\Gamma^\circ]^k \circ (\mathcal{PU})_\Gamma^*$	–	1–7	1–3,5–7
II^+	$[(\mathcal{PU})_\Gamma^* \circ \mathcal{C}_\Gamma^\oplus]^k \circ (\mathcal{PU})_\Gamma^*$	–	1–7	1–3,5–7
II^-	$[(\mathcal{PU})_\Gamma^* \circ \mathcal{C}_\Gamma^\ominus]^k \circ (\mathcal{PU})_\Gamma^*$	–	1–7	1–3,5–7
III^+	$\mathcal{U}_\Gamma \circ \mathcal{P}_\Gamma^* \circ [\mathcal{C}_\Gamma^\oplus]^k$	–	1–5	1–3,5
III^-	$\mathcal{P}_\Gamma^* \circ [\mathcal{C}_\Gamma^\ominus]^k$	–	1–5	1–3,5
IV	$\mathcal{N}_\Gamma \circ [\mathcal{D}_\Gamma^\circ]^k$	$\mathcal{P}_\Gamma$	1–5,8	1–3,5,8
IV^+	$\mathcal{N}_\Gamma \circ [\mathcal{D}_\Gamma^\oplus]^k$	$\mathcal{P}_\Gamma$	1–5,8–9	1–3,5,8–9
V	$\mathcal{N}_\Gamma \circ [\mathcal{P}_\Gamma^* \circ \mathcal{D}_\Gamma^\circ]^k \circ \mathcal{P}_\Gamma^*$	$\mathcal{P}_\Gamma$	1–8	1–3,5–8
V^+	$\mathcal{N}_\Gamma \circ [\mathcal{P}_\Gamma^* \circ \mathcal{D}_\Gamma^\oplus]^k \circ \mathcal{P}_\Gamma^*$	$\mathcal{P}_\Gamma$	1–9	1–3,5–9
VI	$[(\mathcal{PV})_\Gamma^* \circ \mathcal{D}_\Gamma^\circ]^k \circ (\mathcal{PV})_\Gamma^*$	–	1–8	1–3,5–8
VI^+	$[(\mathcal{PV})_\Gamma^* \circ \mathcal{D}_\Gamma^\oplus]^k \circ (\mathcal{PV})_\Gamma^*$	–	1–9	1–3,5–9
VI^-	$[(\mathcal{PV})_\Gamma^* \circ \mathcal{D}_\Gamma^\ominus]^k \circ (\mathcal{PV})_\Gamma^*$	–	1–8	1–3,5–8

The `noMoRe` system

- System architecture
- Coloring strategy and algorithm
- Visualizing color sequences
- Extensions
- Future development

System architecture



System architecture of the **no**n-**mo**notonic **re**asoning system noMoRe.

Computing answer sets

Computation of answer sets (in general) consists of

1. deterministic part: extends a partial (answer set) graph coloring in a reasonable way as much as possible. Only consistent operations are performed. For example, facts have to be in the answer set.

2. non-deterministic part: extends a partial (answer set) graph coloring, which cannot be extended deterministicly, in a systematic (possibly inconsistent) way. For example, a rule $(a \leftarrow not\ a\ )$ must be considered as applied and non-applied.

☞ see definitions of the different operators and results on coloring sequences

☞ the non-deterministic part is subject of heuristics

☞ see the smodels algorithm including expand

The basic coloring strategy of noMoRe

☞ Recall the basic strategy of the noMoRe system:

Let Γ be the *RDG* of logic program Π and let C be a total coloring of Γ.

Then, C is an admissible coloring of Γ

iff

there exists a sequence $(C^i)_{0 \leq i \leq n}$ with the following properties:

1. $C^0 = \mathcal{P}^*_\Gamma((\emptyset, \emptyset))$;
2. $C^{i+1} = \mathcal{P}^*_\Gamma(\mathcal{D}^\circ_\Gamma(C^i))$ where $\circ \in \{\oplus, \ominus\}$ and $0 \leq i < n-1$;
3. $C^n = \mathcal{N}_\Gamma(C^{n-1})$;
4. $C^n = \mathcal{P}_\Gamma(C^n)$;
5. $C^n = C$.

☞ How to implement this strategy?

Implementation: Main algorithm I

☞ code simplified, but very close to the implementation

```
a_color( Col ) :-          % Col is modified during computation
        pre_color(Col,CP),
        !,
        color_cp(CP,Col),
        color_rest(Col).

pre_color( Col, CP ) :-
        facts(Fs),q_facts(QF),
        color_all_1(Fs,Col,QF,CP1),
        loops_1(L1),
        color_all_0(L1,Col,CP1,CP).
```

Implementation: Main algorithm II

```
color_cp( CP, Col ) :-
        heuristic_choose(Node,CP,CP1,Col),!,
        (   Node = no -> true;
            (   cp(Node,Col,[],NewCPs),
                ord_union(NewCPs,CP1,CP2),
                color_cp(CP2,Col)
            )).

cp( Node, Col, CP1, CP2 ) :-
        col_1(Node,Col,CP1,CP2).
cp( Node, Col, CP1, CP2 ) :-
        col_0(Node,Col,CP1,CP2).
```

Implementation: Main algorithm III

☞ Missing predicates

- `color_rest(Col)` is true iff
  all uncolored nodes can be colored $\ominus$ wrt `Col` and we obtain an admissible coloring (see operator-$\mathcal{N}_\Gamma$).

- `color_all_1(Set,Col,InCP,OutCP)` is true iff
  all nodes in `Set` can be colored $\oplus$ wrt `Col` where `OutCP` is `InCP` plus new possible choices (`color_all_0\4` analog).

- `facts\1`, `q_facts\1` and `loop_1\1` give the respective rules.
  Examples
  - `q_facts(h :- not b)` is true (no positive body)
  - `loop_1(h :- a, not, b, not h)` is true (self-blocking)

- `heuristic_choose(Node,InCP,OutCP,Col)` is true iff
  `Node` is some heuristically chosen rule out of `InCP`;
  `OutCP` contains remaining possible choices.

- `col_1\1` and `col_0\1` color a node $\oplus$ or $\ominus$, respectively, and **propagate** the new coloring. If propagation **fails** `col_1/0` **fails**!!

Implementation: Main algorithm IV

```
col_1( Node, Col, CP1, CP2 ) :-
     arg(Node,Col,C),
     ((C == 1) -> CP1=CP2;
         (C == u ->
         (
            setarg(Node,Col,1),
            prop_1(Node,Col,CP1,CP2),
            show(Node,1,Col)
         );fail
     )).
```

☞ How to compute `prop_1/4` and `prop_0/4`?

Implementation: Propagation I

☞ Recall coloring sequence formation: $C^{i+1} = \mathcal{P}^*_\Gamma(\mathcal{D}^\circ_\Gamma(C^i))$ where $\circ \in \{\oplus, \ominus\}$.

- For efficiency operators $\mathcal{P}$ and $\mathcal{D}$ are mixed together in the actual implementation.
- Propagation is performed as local as possible. That is, whenever a node is colored, its color is propagated to the direct successors of that node.
- No need for global computation of $S(\Gamma, C)$, $\overline{S}(\Gamma, C)$, $B(\Gamma, C)$ and $\overline{B}(\Gamma, C)$.
- Next possible choices are computed online during propagation.
- Propagation is performed recursively until no more nodes can be colored by propagation.

Implementation: Propagation II

Let C be a partial coloring of Γ_Π (Π logic program) and let u $\in \Pi$ be some uncolored node wrt C. We have the following (local) forward propagation cases:

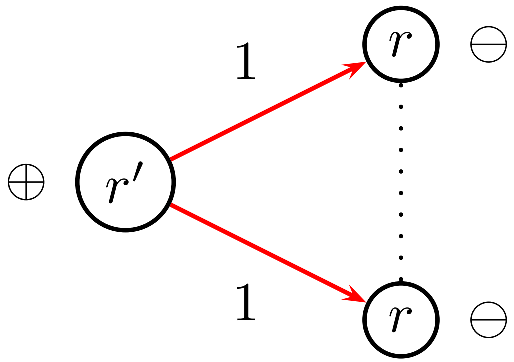
1. **prop**$(\oplus \overset{1}{\to}$ u$) \mapsto (\oplus \overset{1}{\to} \ominus)$ no condition
2. **prop**$(\oplus \overset{0}{\to}$ u$) \mapsto (\oplus \overset{0}{\to} \oplus)$ if u $\in S(\Gamma, C) \cup \overline{B}(\Gamma, C)$
3. **prop**$(\ominus \overset{1}{\to}$ u$) \mapsto (\ominus \overset{1}{\to} \oplus)$ if u $\in S(\Gamma, C) \cup \overline{B}(\Gamma, C)$
4. **prop**$(\ominus \overset{0}{\to}$ u$) \mapsto (\ominus \overset{0}{\to} \ominus)$ if u $\in \overline{S}(\Gamma, C)$

☞ `prop_1\4` calls cases 1. and 2. whereas `prop_0\4` calls cases 3. and 4.

☞ if **prop**$(\oplus \overset{0}{\to}$ u$) \mapsto (\oplus \overset{0}{\to} \oplus)$ fails since u $\in S(\Gamma, C)$ but u $\notin \overline{B}(\Gamma, C)$, then u is collected as a new choice.

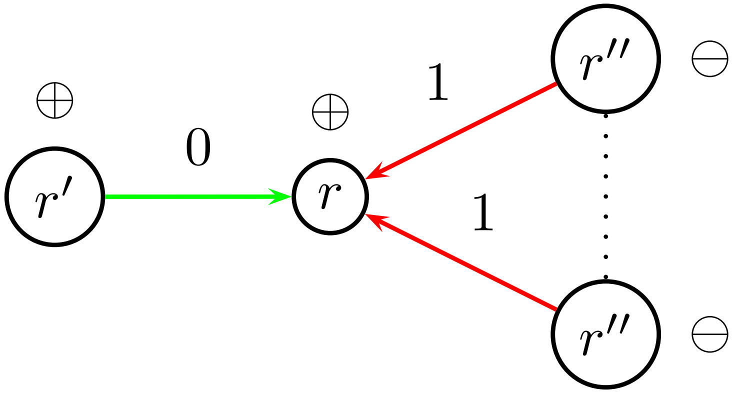
Propagation: $\mathbf{prop}(\oplus \xrightarrow{1} \mathbf{u}) \mapsto (\oplus \xrightarrow{1} \ominus)$

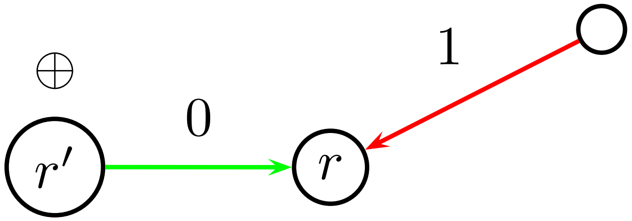
Condition: no



Propagation: $\mathbf{prop}(\oplus \xrightarrow{0} \mathbf{u}) \mapsto (\oplus \xrightarrow{0} \oplus)$

Condition: $r \in S(\Gamma, C) \cup \overline{B}(\Gamma, C)$

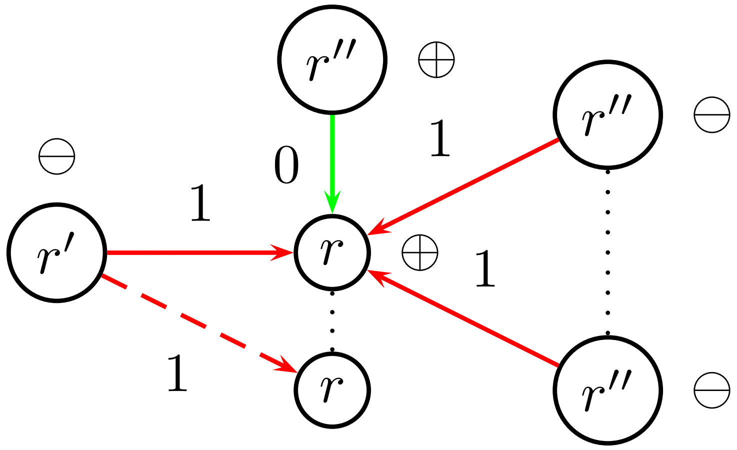




r **is new possible choice!!**

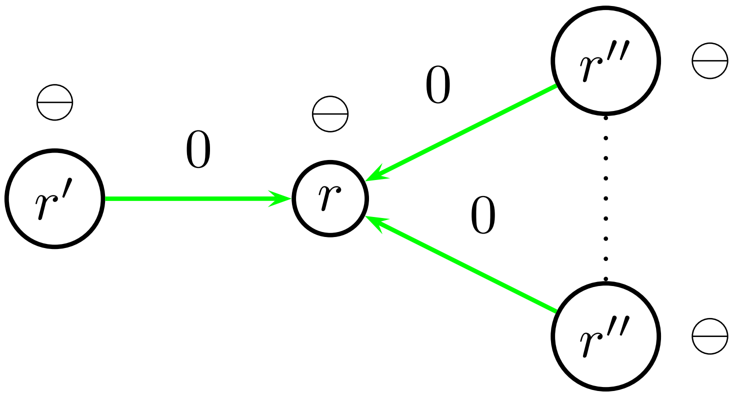
Propagation: $\mathbf{prop}(\ominus \xrightarrow{1} \mathbf{u}) \mapsto (\ominus \xrightarrow{1} \oplus)$

Condition: $r \in S(\Gamma, C) \cup \overline{B}(\Gamma, C)$



Propagation: $\textbf{prop}(\ominus \xrightarrow{0} \mathbf{u}) \mapsto (\ominus \xrightarrow{0} \ominus)$

Condition: $r \in \overline{S}(\Gamma, C)$



Implementation: Backward propagation I

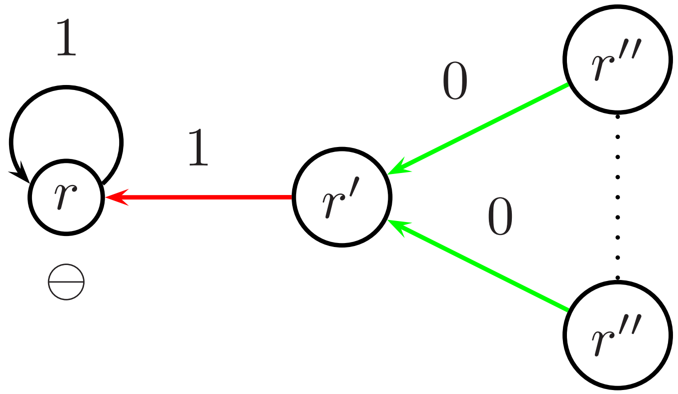
Let C be a partial coloring of Γ_Π (Π logic program) and let u $\in \Pi$ be some uncolored node wrt C. We have the following (local) backward propagation cases:

1. **bprop**$(\oplus \stackrel{1}{\leftarrow} \mathrm{u}) \mapsto (\oplus \stackrel{1}{\leftarrow} \ominus)$ no condition
2. **bprop**$(\oplus \stackrel{0}{\leftarrow} \mathrm{u}) \mapsto (\oplus \stackrel{0}{\leftarrow} \oplus)$ if first node is in $\overline{B}(\Gamma, C)$ and only u is responsible for supporting the first one
3. **bprop**$(\ominus \stackrel{1}{\leftarrow} \mathrm{u}) \mapsto (\ominus \stackrel{1}{\leftarrow} \oplus)$ if first node is in $S(\Gamma, C)$ and only u is responsible for blocking the first one
4. **bprop**$(\ominus \stackrel{0}{\leftarrow} \mathrm{u}) \mapsto (\ominus \stackrel{0}{\leftarrow} \ominus)$ if first node is in $\overline{B}(\Gamma, C)$ and only u is responsible for non-supporting the first one

☞ backward propagation is only necessary for choices and 1-loops and it works recursively.

Backward propagation II

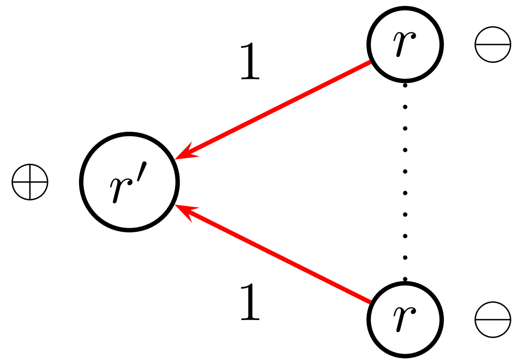
☞ For backward propagation we have to look at partial mappings $C : \Pi \mapsto \{\ominus, \oplus, +\}$ for graph coloring.



☞ All $+$ colored nodes have to be colored $\oplus$ in order to end with an admissible coloring.

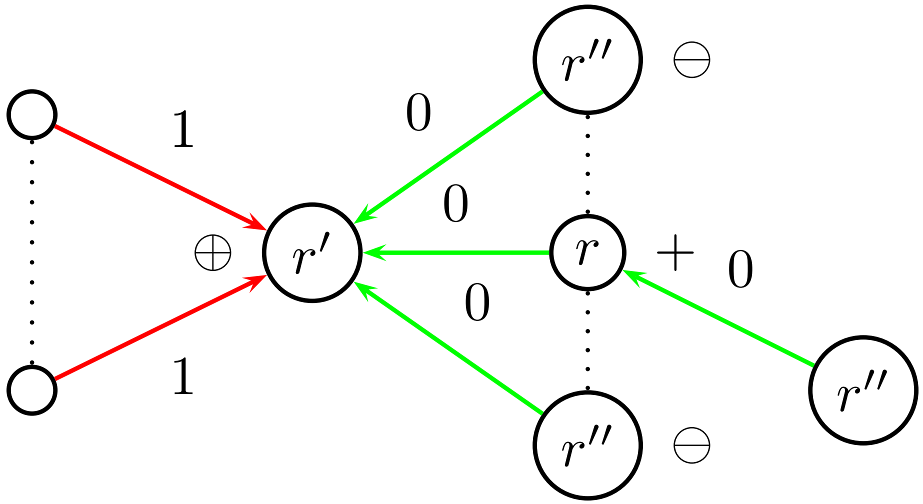
Backward propagation: bprop$(\oplus \xleftarrow{1} \mathbf{u}) \mapsto (\oplus \xleftarrow{1} \ominus)$

Condition: no



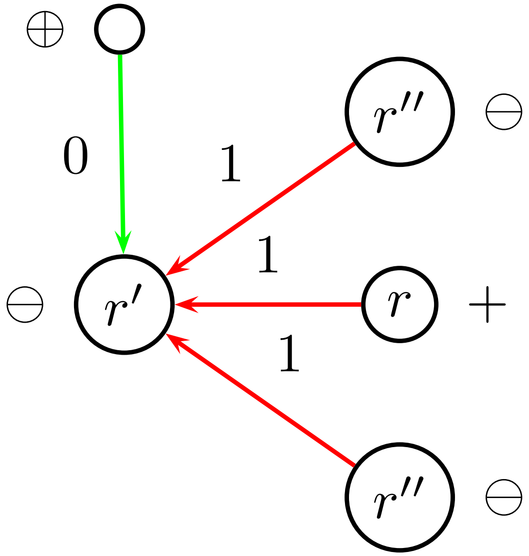
Backward propagation: bprop$(\oplus \xleftarrow{0} \mathbf{u}) \mapsto (\oplus \xleftarrow{0} \oplus)$

Condition: $r' \in \overline{B}(\Gamma, C)$



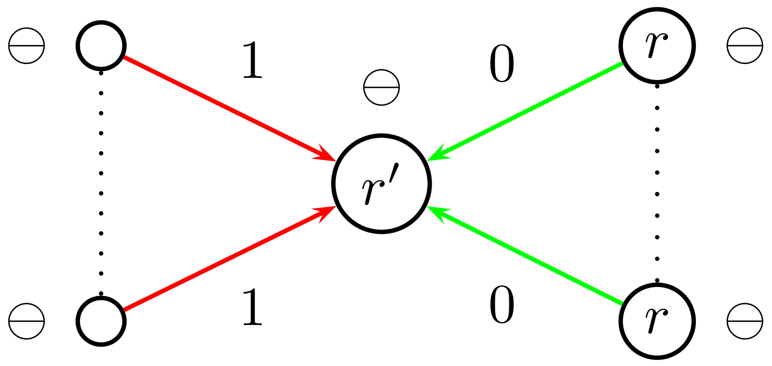
Backward propagation: bprop$(\ominus \xleftarrow{1} \mathbf{u}) \mapsto (\ominus \xleftarrow{1} \oplus)$

Condition: $r' \in S(\Gamma, C)$



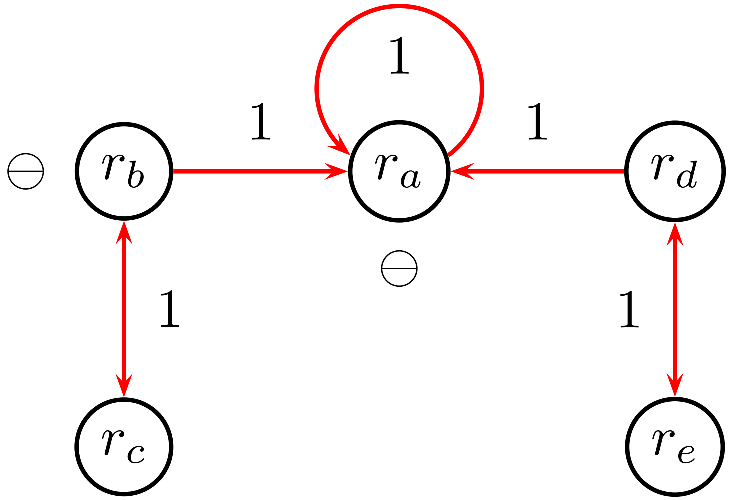
Backward propagation: bprop$(\ominus \xleftarrow{0} \mathbf{u}) \mapsto (\ominus \xleftarrow{0} \ominus)$

Condition: $r' \in \overline{B}(\Gamma, C)$



Backward propagation: Jumping

$$
\begin{aligned}
r_a &= a \leftarrow not\ \ a, not\ \ b, not\ \ d \\
r_b &= b \leftarrow not\ \ c \\
r_c &= c \leftarrow not\ \ b \\
r_d &= d \leftarrow not\ \ e \\
r_e &= e \leftarrow not\ \ d
\end{aligned}
$$



Backward propagation: Experiments

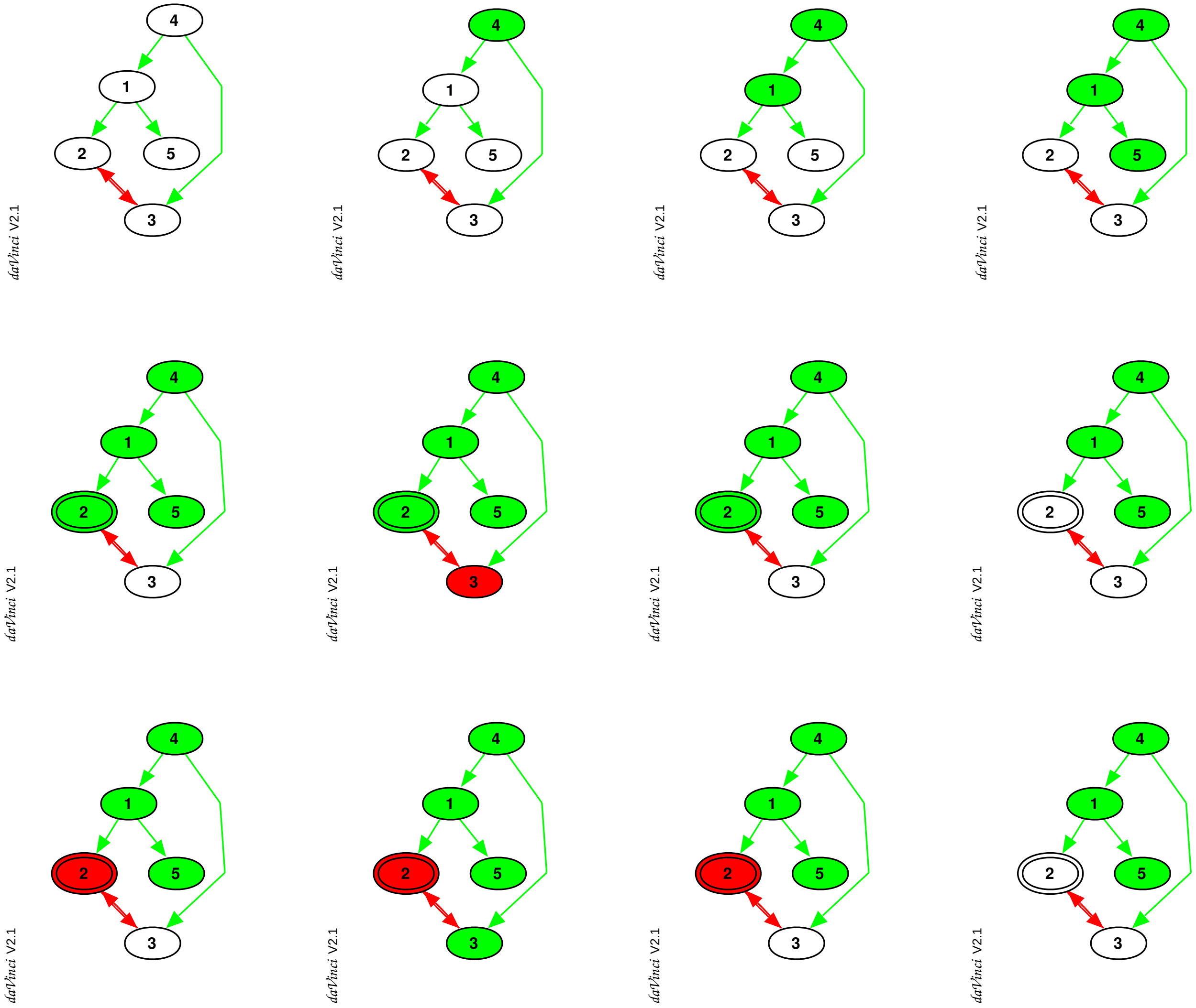
	noMoRe			smodels	
backprop	no	yes	yes	with	(without)
jumping	no	no	yes	lookahead	
ham_k_7	14335	14335	2945	4814	(34077)
ham_k_8	82200	82200	24240	688595	(86364)
ind_cir_20	539	317	276 $*$	276	(276)
ind_cir_30	9266	5264	4609 $*$	4609	(4609)
p1_step4	-	464	176	7	(69)
p2_step6	-	13654	3779	75	(3700)
col4x4	27680	27680	7811	7811	(102226)
col5x5	-	-	580985	580985	(2.3 Mil)
queens4	84	84	5	1	(11)
queens5	326	326	13	9	(34)

Visualizing color sequences: An example

☞ noMoRe has an interface to the graph drawing tool daVinci, which gives animated coloring sequences

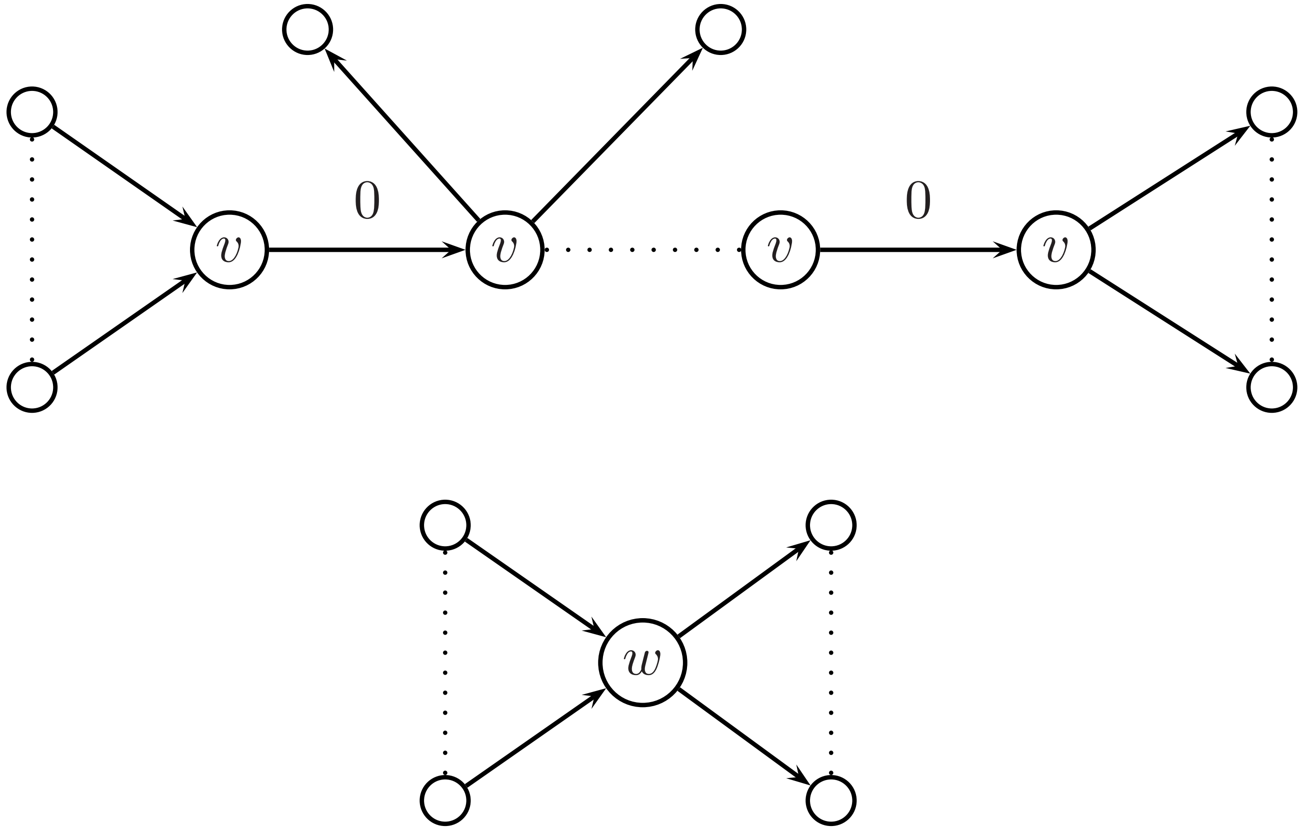
$$\begin{array}{lcccl} r_1 & : & b & \leftarrow & p \\ r_2 & : & f & \leftarrow & b, not \ \ f' \\ r_3 & : & f' & \leftarrow & p, not \ \ f \\ r_4 & : & p & \leftarrow & \\ r_5 & : & w & \leftarrow & b \end{array}$$

Visualizing color sequences: An example



Extension: Transformations

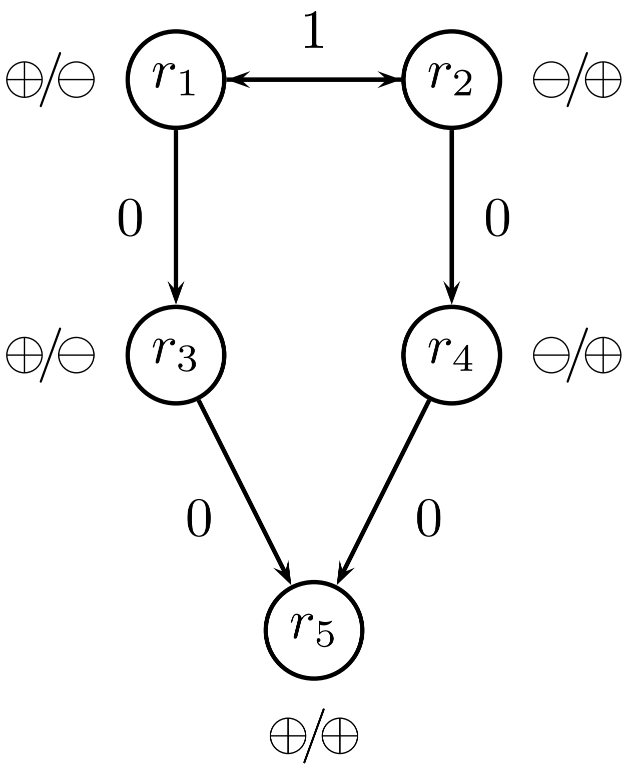
☞ for dependency graphs

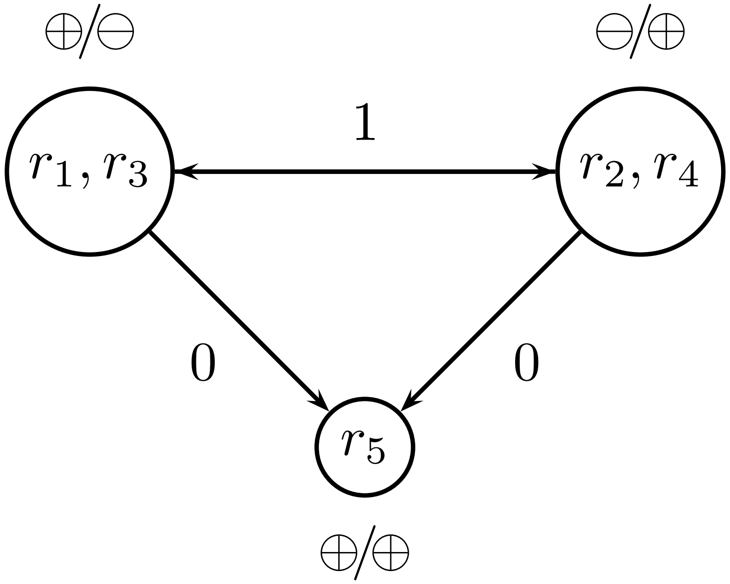


Transformations: An example

$$r_1 : a \leftarrow not\ b. \qquad r_2 : b \leftarrow not\ a.$$

$$r_3 : c \leftarrow a. \qquad r_4 : c \leftarrow b. \qquad r_5 : d \leftarrow c.$$





☞ no normal program with same answer sets corresponds to the transformed graph.

Extension: Syntax

By suitable generalization of RDGs noMoRe is able to deal with (ground) normal nested logic programs, that is, rule of the form

$$h_1, \ldots, h_k \leftarrow B_1; \ldots; B_n \qquad (1)$$

where each B_i is a conjunction of default literals ("normal" body).

- $p(a)$ and $q(a,b)$ are propositional atoms
- conjunction and disjunction are allowed as in rule (1), for example
  - a normal rule $p(a) \leftarrow b, not\ \ c$ is represented by `p(a) :- b, not c.`
  - a nested rule $a, b \leftarrow (c, not\ \ d); (e)$ is represented by `a,b :- (c, not d);(e).`
  - a integrity constraint $\leftarrow p, q$ is represented by `:- p, q.`
- lines beginning with a % are regarded as a comment lines

Extension: Transformations

☞ for logic programs

- all rules $\Pi_\varphi \subseteq \Pi$ with head φ are transformed to one rule $\varphi \leftarrow B_1; \ldots; B_n$ where $body(\Pi_\varphi) = \{B_1, \ldots, B_n\}$.

- all rules $\Pi_\phi \subseteq \Pi$ with body ϕ are transformed to one rule $h_1, \ldots, h_n \leftarrow \phi$ where $head(\Pi_\phi) = \{h_1, \ldots, h_n\}$.

☞ Observe, that this program transformations can be applied since noMoRe is able to deal with normal nested programs.

noMoRe ++

- C++ implementation

Extensions and future development

- heuristics for choices
- more transformations
- preferences between rules
- extension to disjunctive programs

Classical Negation $\neg$

- Motivation
- Semantics
- Translation

Motivation

- Given a set X, the difference among $not\ a$ and $\neg a$ amounts to:

$$a \notin X \qquad \text{versus} \qquad \neg a \in X$$

- Example

$$cross \leftarrow\ not\ train \qquad\qquad cross \leftarrow\ \neg\ train$$

$$X = \{cross\} \qquad\qquad X = \emptyset$$

- Two- versus Three-valued-interpretation of answer sets

  For instance, given $\mathcal{A} = \{a, b\}$ and $X = \{a\}$,

  - b is false under a two-valued interpretation of X and
  - b is undefined under a three-valued interpretation of X.

Literals

- A *literal* L is an atom A or its negation $\neg A$.
- Two literals A and $\neg A$ are said to be *complementary*.
- A set of literals is *inconsistent*, if it contains a complementary pair of literals, and *consistent* otherwise.
- A set of literals is *logically closed* if it is consistent or if it equals the set of all literals.

Extended logic programs

- An *extended rule*, r, is an ordered pair of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n,$$

  where $n \geq m \geq 0$, and each L_i $(0 \leq i \leq n)$ is a *literal*.

- An *extended logic program* is a finite set of extended rules.

- Notations

$$\begin{aligned} head(r) &= L_0 \\ body(r) &= \{L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\} \\ body^+(r) &= \{L_1, \ldots, L_m\} \\ body^-(r) &= \{L_{m+1}, \ldots, L_n\} \end{aligned}$$

- A program is called *basic* if $body^-(r) = \emptyset$ for all its rules.

Answer sets

- The smallest set of atoms which is logically closed *and* closed under a basic program Π is denoted by $Cn(\Pi)$.
- A set X of atoms is an *answer set* of a program Π if $Cn(\Pi^X) = X$.

Back to normal logic programs

- Define $\mathcal{A}' = \{A' \mid A \in \mathcal{A}\}$.
- The mapping *norm* between extended and normal logic programs is defined recursively as follows:
  - $norm(A) = A$ if A is an atom,
  - $norm(\neg A) = A'$ if A is an atom,
  - $norm(not\ L) = not\ norm(L)$ if L is a literal,
  - $norm(X) = \{norm(L) \mid L \in X\}$ if X is a set,
  - $norm(\Pi) = \{norm(head(r)) \leftarrow norm(body(r)) \mid r \in \Pi\}$

  plus the set $\Pi_\perp$ of contradiction rules
  - $A \leftarrow B, B'$
  - $A' \leftarrow B, B'$

  for all distinct pairs A, B of atoms in $\mathcal{A}$.

Properties of norm

- For any basic program Π,

$$norm(Cn(\Pi)) = Cn(norm(\Pi) \cup \Pi_\perp)$$

- Let Π be an extended logic program and let X be a set of literals. Then, we have that

  X is an answer set of Π iff $norm(X)$ is an answer set of $norm(\Pi) \cup \Pi_\perp$.

- Let Π be an extended logic program and let X be a *consistent* set of literals. Then, we have that

  X is an answer set of Π iff $norm(X)$ is an answer set of $norm(\Pi)$.

An example

$$\Pi = \left\{ \begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ \neg p \end{array} \right\} \qquad norm(\Pi) \cup \Pi_\perp = \left\{ \begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ p' \end{array} \right\} \cup \left\{ \begin{array}{l} p \leftarrow q, q' \\ p' \leftarrow q, q' \\ q \leftarrow p, p' \\ q' \leftarrow p, p' \end{array} \right\}$$

- Π has answer set $\{q, \neg p\}$
- $norm(\Pi) \cup \Pi_\perp$ has answer set $\{q, p'\}$
- $norm(\Pi)$ has two answer sets $\{q, p'\}$ and $\{p, p'\}$

Preferences

- Motivation
- Semantics
- Compilation

Motivation

The notion of *preference* in commonsense reasoning is pervasive.

For instance,

- in buying a car, one may prefer certain features over others;
- in scheduling, meeting some deadlines may be more important than meeting others;
- in legal reasoning, laws are subject to higher principles, like lex superior or lex posterior, which are themselves subject to “higher higher” principles;
- etc etc . . .

Legal reasoning

The challenge!

“A person wants to find out if her security interest in a certain ship is perfected. She currently has possession of the ship. According to the Uniform Commercial Code (UCC, §9-305) a security interest in goods may be perfected by taking possession of the collateral. However, there is a federal law called the Ship Mortgage Act (SMA) according to which a security interest in a ship may only be perfected by filing a financing statement. Such a statement has not been filed. Now the question is whether the UCC or the SMA takes precedence in this case. There are two known legal principles for resolving conflicts of this kind. The principle of Lex Posterior gives precedence to newer laws. In our case the UCC is newer than the SMA. On the other hand, the principle of Lex Superior gives precedence to laws supported by the higher authority. In our case the SMA has higher authority since it is federal law.”

(Gordon, 1993)

Legal reasoning

Our solution in *"ordered logic programming"*

```
    perfected :- name(ucc), possession,             not neg perfected.
neg perfected :- name(sma), ship, neg finstatement, not     perfected.

possession.   ship.   neg finstatement.

(Y < X) :- name(lex_posterior(X,Y)), newer(X,Y), not neg (Y < X).

(X < Y) :- name(lex_superior(X,Y)), state_law(X), federal_law(Y), not neg (X < Y).

newer(ucc,sma).   federal_law(sma).   state_law(ucc).

(lex_posterior(X,Y) < lex_superior(X,Y)).
```

What type of preference?

Different (often dependent) options:

- Preferences on rules
  (versus preferences on literals
  or preferences on answer sets)
- Selection function on the set of answer sets
  (versus possible modification of answer sets)
- Complexity within NP
  (versus complexity beyond NP)

How to express preferences?

Static preferences: Use an external order $<$.

Ordered logic program: $(\Pi, <)$

where Π is a logic program over $\mathcal{L}$ and
$<$ is a strict partial order over Π ;

Dynamic preferences: Use a special-purpose predicate $\prec$.

Ordered logic program: Π

where Π is a logic program over $\mathcal{L} \cup \{\prec\}$ containing
rules expressing that $\prec$ is a strict partial order.

Every statically ordered program can be expressed as a dynamically ordered one.

An example

Consider the following ordered logic program $(\Pi, <)$ with $\Pi = \{r_1, r_2, r_3\}$

$$\begin{array}{rrcl} r_1: & \neg a & \leftarrow & \\ r_2: & b & \leftarrow & \neg a, not\ c \\ r_3: & c & \leftarrow & not\ b \end{array} \qquad \text{and} \qquad r_3 < r_2\ .$$

This program has two standard answer sets,

$\{\neg a, b\}$ and $\{\neg a, c\}$

among which the green one is preferred.

How to define preferred answer sets?

for *statically* ordered logic programs

Let $(\Pi, <)$ be an ordered program and let X be an answer set of Π.

Then, X is $<$-*preserving*, if X is either inconsistent, or else there exists an enumeration $\langle r_i \rangle_{i \in I}$ of $R_\Pi(X)$ such that for all $i, j \in I$ we have that:

1. $body^+(r_i) \subseteq \{head(r_j) \mid j < i\}$; and
2. if $r_i < r_j$, then $j < i$; and
3. if $r_i < r'$ and $r' \in \Pi \setminus R_\Pi(X)$, then
   (a) $body^+(r') \not\subseteq X$ or
   (b) $body^-(r') \cap \{head(r_j) \mid j < i\} \neq \emptyset$.

Recall: $R_\Pi(X) = \{r \in \Pi \mid body^+(r) \subseteq X \text{ and } body^-(r) \cap X = \emptyset\}$.

An example

Consider the statically ordered logic program

$$(\Pi, <) = (\{r_1, r_2, r_3\}, \{r_3 < r_2\})$$

with

$$\begin{array}{rcrcl} r_1 & = & \neg a & \leftarrow & \\ r_2 & = & b & \leftarrow & \neg a, not\ c \\ r_3 & = & c & \leftarrow & not\ b \end{array}$$

The (standard) program Π has two answer sets:

1. $\{\neg a, b\}$ is $<$-preserving; $\langle r_1, r_2 \rangle$ satisfies (1) and (2).
2. $\{\neg a, c\}$ is not $<$-preserving, since $\langle r_1, r_3 \rangle$ violates (3b).

Implementation

for *dynamically* ordered logic programs

Idea

Translate a logic program Π with preference information into a regular logic program $\mathcal{T}(\Pi)$ such that answers to $\mathcal{T}(\Pi)$ respect the preferences in Π.

Plan

1. Extend the language for expressing preference
2. Add axioms encoding specific preference handling strategies

(Dynamically) ordered logic programs

An *ordered logic program* is an extended logic program over a propositional language $\mathcal{L}$,

containing the following pairwise disjoint categories:

- a set N of terms serving as *names* for rules;
- a set $\mathcal{A}$ of regular (propositional) atoms of a program; and
- a set $\mathcal{A}_\prec$ of *preference atoms* $s \prec t$, where $s, t \in N$ are names.

For each ordered program Π, we require a bijective function $n(\cdot)$ assigning to each rule $r \in \Pi$ a name $n(r) \in N$.

To simplify our notation, we write

- n_r instead of $n(r)$ or n_i instead of n_{r_i} and
- $t : r$ instead of $t = n(r)$.

Towards preferred answer sets

1. Introduce special purpose predicates controlling rule application:

   $\mathsf{ap}(n_r)$ signifies that rule r is applicable wrt X, that is,

   - $body^+(r) \subseteq X$ and
   - $body^-(r) \cap X = \emptyset$.

   $\mathsf{bl}(n_r)$ signifies that rule r is blocked wrt X, that is, either

   - $body^+(r) \not\subseteq X$ or
   - $body^-(r) \cap X \neq \emptyset$.

   $\mathsf{ok}(n_r)$ signifies that it is "ok" to consider rule r

2. Provide axioms that guarantee a consideration of rules that is in accord with the underlying preference information, that is,

   $n_r \prec n_{r'}$ enforces that $\mathsf{ok}(n_{r'})$ is derivable "before" $\mathsf{ok}(n_r)$

3. Specify what it means that a rule "has been considered"

Translating ordered logic programs

Let $\Pi = \{r_1, \ldots, r_k\}$ be an ordered logic program over $\mathcal{L}$.

Let $\mathcal{L}^\star$ be the language obtained from $\mathcal{L}$ by adding, for each $r, r' \in \Pi$, new pairwise distinct propositional atoms

- $\mathsf{ap}(n_r)$,
- $\mathsf{bl}(n_r)$,
- $\mathsf{ok}(n_r)$, and
- $\mathsf{rdy}(n_r, n_{r'})$.

Then, the logic program $\mathcal{T}(\Pi)$ over $\mathcal{L}^\star$ contains the following rules,
`(shown on the next slide)`

Translating ordered logic programs (ctd)

For each $r \in \Pi$, where $L^+ \in body^+(r)$, $L^- \in body^-(r)$, and $r', r'' \in \Pi$:

$$
\begin{array}{rrcl}
a_1(r): & head(r) & \leftarrow & \mathsf{ap}(n_r) \\
a_2(r): & \mathsf{ap}(n_r) & \leftarrow & \mathsf{ok}(n_r), body(r) \\
b_1(r,L): & \mathsf{bl}(n_r) & \leftarrow & \mathsf{ok}(n_r), not\ L^+ \\
b_2(r,L): & \mathsf{bl}(n_r) & \leftarrow & \mathsf{ok}(n_r), L^- \\
\\
c_1(r): & \mathsf{ok}(n_r) & \leftarrow & \mathsf{rdy}(n_r, n_{r_1}), \ldots, \mathsf{rdy}(n_r, n_{r_k}) \\
c_2(r,r'): & \mathsf{rdy}(n_r, n_{r'}) & \leftarrow & not(n_r \prec n_{r'}) \\
c_3(r,r'): & \mathsf{rdy}(n_r, n_{r'}) & \leftarrow & (n_r \prec n_{r'}), \mathsf{ap}(n_{r'}) \\
c_4(r,r'): & \mathsf{rdy}(n_r, n_{r'}) & \leftarrow & (n_r \prec n_{r'}), \mathsf{bl}(n_{r'}) \\
\\
t(r,r',r''): & n_r \prec n_{r''} & \leftarrow & n_r \prec n_{r'}, n_{r'} \prec n_{r''} \\
as(r,r'): & \neg(n_{r'} \prec n_r) & \leftarrow & n_r \prec n_{r'}
\end{array}
$$

An(other) example

Consider the following ordered logic program $\Pi = \{r_1, r_2, r_3, r_4\}$:

$$\begin{array}{lcrcl} r_1 & = & \neg a & \leftarrow & \\ r_2 & = & b & \leftarrow & \neg a, not\ c \\ r_3 & = & c & \leftarrow & not\ b \\ r_4 & = & n_3 \prec n_2 & \leftarrow & not\ d \end{array}$$

where n_i denotes the name of rule r_i $(i = 1, \ldots, 4)$.

This program has two answer sets, $\{\neg a, b, n_3 \prec n_2\}$ and $\{\neg a, c, n_3 \prec n_2\}$.

Only the first one, $\{\neg a, b, n_3 \prec n_2\}$, preserves its contained preference, given by $n_3 \prec n_2$.

Properties

Let $(\Pi, <)$ be an ordered logic program and X a set of literals.

Soundness and correctness

X is a $<$-preserving answer set of Π

iff

$X = Y \cap \mathcal{L}$ for some answer set Y of $\mathcal{T}(\Pi \cup \{(n_r \prec n_{r'}) \leftarrow \mid r < r'\})$.

➥ One-to-one correspondence (actually holds).

Selection

If Y is an answer set of $\mathcal{T}(\Pi \cup \{(n_r \prec n_{r'}) \leftarrow \mid r < r'\})$,
then $Y \cap \mathcal{L}$ is an answer set of Π.

Implementation

plp `http://www.cs.uni-potsdam.de/~torsten/plp`

- Front-end to `dlv` and `smodels`
- Ordered logic programs
  - with preferences (everywhere)

    eg. $n_{17} \prec n_{42} \leftarrow \ n_{17} \prec n_{34}, \ \mathit{not} \ (n_{42} \prec n_{34})$
  - with variables

    eg. $n_1(x) \prec n_2(y) \leftarrow \ p(y), \ \mathit{not} \ (x = c)$
  - with disjunctive preferences

    eg. $(r_2 \prec r_{42}) \vee (r_4 \prec r_{42}) \leftarrow \neg a$

The example ran through plp

Ordered logic program $\Pi = \{r_1, r_2, r_3, r_4\}$:

$$
\begin{aligned}
r_1 &= & \neg a &\leftarrow \\
r_2 &= & b &\leftarrow \neg a, not\ c \\
r_3 &= & c &\leftarrow not\ b \\
r_4 &= & n_3 \prec n_2 &\leftarrow not\ d
\end{aligned}
$$

becomes

```
neg a.
b :- name(n2), neg a, not c.
c :- name(n3), not b.
(n3 < n2) :- not d.
```

The outcome

```
neg_a.
b :- ap(n2).
ap(n2) :- ok(n2), neg_a, not c.
bl(n2) :- ok(n2), not neg_a.
bl(n2) :- ok(n2), c.
c :- ap(n3).
ap(n3) :- ok(n3), not b.
bl(n3) :- ok(n3), b.
prec(n3, n2) :- not d.
ok(N) :- name(N), rdy(N, n2), rdy(N, n3).
rdy(N, M) :- name(N), name(M), not prec(N, M).
rdy(N, M) :- name(N), name(M), prec(N, M), ap(M).
rdy(N, M) :- name(N), name(M), prec(N, M), bl(M).
neg_prec(M, N) :- name(N), name(M), prec(N, M).
prec(N, M) :- name(N), name(M), name(O),
              prec(N, O), prec(O, M).
false :- a, neg_a. false :- b, neg_b. false :- c, neg_c. false :- d, neg_d.
false :- name(N), name(M), prec(N, M), neg_prec(N, M).
name(n3). name(n2).
```

Computing preferred answer sets

```
?- lp2dlv('Examples/example').

yes
?- dlv('Examples/example').
dlv [build BEN/Apr  5 2000   gcc 2.95.2 19991024 (release)]

{name(n2), name(n3), neg_a, ok(n2), rdy(n2,n2), rdy(n2,n3), rdy(n3,n3),
 prec(n3,n2), neg_prec(n2,n3), ap(n2), b, rdy(n3,n2), ok(n3), bl(n3)}

yes
?- dlv('Examples/example',nice).
dlv [build BEN/Apr  5 2000   gcc 2.95.2 19991024 (release)]

{neg_a, b}

yes
?-
```

Sokoban - Initial Information

- `initial_at(X,Y)`
- `square(X,Y)`
- `initial_box(X,Y)`
- `target_square(X,Y)`

Sokoban - Beginning and End

```
time(1..n).

at(X, Y, 1) :-
    initial_at(X, Y).

has_box(X, Y, 1) :-
    initial_box(X, Y).

% In the end, all target squares must have boxes:
compute { has_box(X, Y, n +1) : target_square(X, Y) }.
```

Sokoban - Has Box I

```
% A box ends where it is pushed to
has_box(X, Y, I+1) :-
    move_to(X, Y, I),
    move_square(X, Y),
    time(I).

% A box moves away when pushed
-has_box(X, Y, I+1) :-
    push(X, Y, Dir, I),
    time(I),
    has_neighbor(X, Y, Dir).
```

Sokoban - Has Box II

```
% A box stays at a place if not pushed
has_box(X, Y, I+1) :-
    not -has_box(X, Y, I+1),
    has_box(X, Y, I),
    time(I),
    move_square(X, Y).

% A box may not be pushed onto another
 :- has_box(X, Y, I),
    move_to(X, Y, I),
    move_square(X, Y),
    time(I).
```

Sokoban - Has Box III

```
% A box may not be pushed over another
 :- has_box(X_2, Y_2, I),
    push(X_1, Y_1, Dir, I),
    move_to(X_3, Y_3, I),
    time(I),
    same_segment(X_1, Y_1, X_2, Y_2, Dir),
    same_segment(X_2, Y_2, X_3, Y_3, Dir).
```

Sokoban - Move to

```
% A box ends at exactly one position along the push direction
1 { move_to(X_2, Y_2, I) : same_segment(X_1, Y_1, X_2, Y_2, Dir) } 1 :-
    push(X_1, Y_1, Dir, I),
    has_neighbor(X_1, Y_1, Dir),
    time(I), I < n.
```

Sokoban - Pushing Boxes I

```
% A box may be pushed if it can be pushed:
{ push(X, Y, Dir, I) } :-
    can_push(X, Y, Dir, I),
    has_box(X, Y, I),
    has_neighbor(X, Y, Dir),
    possible_box(X, Y, I),
    time(I).

% No two boxes may be pushed at one time
 :- 2 { push(X, Y, Dir, I) :
           move_square(X, Y) :
           direction(Dir) },
    time(I).
```

Sokoban - Pushing Boxes II

```
% A box can be pushed in a direction if the worker can reach it and
% there is space immediately behind the box
can_push(X, Y, east, I) :-
    has_box(X, Y, I),
    move_square(X, Y ;  X+1, Y),
    square(X-1, Y),
    not has_box(X-1, Y, I),
    not has_box(X+1, Y, I),
    reachable(X-1, Y, I),
    possible_box(X, Y, I),
    time(I).
```

Sokoban - Moving Boxes

```
% We only try to move to places from where there is a route to a
% target location:
move_square(X, Y) :-
    square(X, Y),
    has_target_route(X, Y).

has_target_route(X, Y) :-
    target_square(X, Y).

has_target_route(X, Y) :-
    square(X, Y ; X+1, Y; X-1, Y),
    has_target_route(X+1, Y).

usw.
```

Sokoban - Reachable

```
% The worker can reach all places that are not blocked.
reachable(X, Y, I) :-
    square(X, Y),
    time(I),
    at(X, Y, I).

reachable(X+1, Y, I) :-
    square(X, Y ; X+1, Y),
    time(I),
    reachable(X, Y, I),
    not has_box(X+1, Y, I).

usw.
```

Sokoban - Has Neighbor

```
% When squares have neighbors:
has_neighbor(X, Y, east) :-
    move_square(X, Y ; X +1, Y).
usw.
```

Sokoban - Improvements I

```
% Check if goal is reached:
goal(I) :-
    { not has_box(X, Y, I) : target_square(X, Y) } 0,
    time(I).

% Do not push after goal is reached:
 :- 1 { move_to(X, Y, I) : move_square(X, Y) },
    time(I),
    goal(I).
```

Sokoban - Improvements II

```
% The final move has to be to a target square:
1 { move_to(X_2, Y_2, n) : same_segment(X_1, Y_1, X_2, Y_2, Dir) :
    target_square(X_2, Y_2) } 1 :-
    push(X_1, Y_1, Dir, n),
    has_neighbor(X_1, Y_1, Dir).

% A box may not be pushed twice to the same direction:
 :- push_dir(Dir, I),
    move_to(X, Y, I),
    push(X, Y, Dir, I),
    has_neighbor(X, Y, Dir),
    time(I).
```

Sokoban - Improvements III

```
% no immediate undoing of a move, if reachable doesn't change
 :- push(X, Y, west, I),
    push_dir(east, I),
    move_to(X, Y, I+1),
    time(I), I < n,
    has_neighbor(X, Y, west),
    reachable(X-2, Y, I).

usw.
```

Sokoban - Improvements IV

```
% Don't push two boxes adjacent each other along the edge.
 :- edge_pair(X_1, Y_1, X_2, Y_2),
    time(I),
    has_box(X_1, Y_1, I),
    has_box(X_2, Y_2, I).

edge_pair(X, Y, X+1, Y) :-
    move_square(X, Y ; X+1, Y),
    not target_square(X, Y),
    not target_square(X+1, Y),
    not square(X, Y-1),
    not square(X+1, Y-1).

usw.
```

Sokoban - Improvements V

```
% Don't push three boxes into a L-turn:
 :- { target_square(X_1, Y_1),
      target_square(X_2, Y_2),
      target_square(X_3, Y_3) } 2,
    l_turn(X_1, Y_1, X_2, Y_2, X_3, Y_3),
    time(I),
    has_box(X_1, Y_1,I),
    has_box(X_2, Y_2,I),
    has_box(X_3, Y_3,I).

l_turn(X,Y, X+1, Y, X+1, Y+1) :-
    move_square(X, Y ; X+1, Y ; X+1, Y+1),
    not square(X+1, Y).
```

Sokoban - Improvements VI

```
% Don't form a 4-box square:
 :- move_square(X, Y),
    move_square(X+1,Y),
    move_square(X+1, Y+1),
    move_square(X, Y+1),
    time(I),
    not target_square(X, Y),
    not target_square(X+1,Y),
    not target_square(X+1, Y+1),
    not target_square(X, Y+1),
    has_box(X, Y, I),
    has_box(X+1, Y, I),
    has_box(X+1, Y+1, I),
    has_box(X, Y+1, I).
```

Sokoban - Improvements VII

```
% Treat first two moves special because we know what moves are possible:
possible_box(X, Y, 1) :- initial_box(X, Y).

possible_box(X_2, Y_2, 2) :-
    same_segment(X_1, Y_1, X_2, Y_2, Dir),
    move_square(X_2, Y_2),
    initial_box(X_1, Y_1).

possible_box(X, Y, 2) :-
    initial_box(X, Y).

possible_box(X, Y, I) :-
    time(I), I >= 3,
    move_square(X, Y).
```

What is a configuration problem?

Product: consists of different components *(configuration objects)* that interact in complex ways

Configuration model: collection of objects and relationships between them

Constraints: restrict allowed object combinations in the model

Configuration: set of objects of the configuration model

Configuration process: find a configuration that satisfies given user requirements in a given configuration model

Classes of configurations

Valid configuration: satisfies all constraints of the configuration model

Suitable configuration: is a valid configuration satisfying all user requirements

Optimal configuration: is a suitable configuration satisfying some optimal criteria

Debian GNU/Linux System

The Debian configuration model can be divided in two parts:

1. a *database* that stores information about the packages and their relations as facts;
2. a set of *inference rules* that construct the valid configurations using the facts stored in the database.

Database

package(P) software package P

depends(P$_1$,P$_2$) package P_1 depends on package P_2 (package P_1 cannot be used if P_2 is not installed)

conflicts(P$_1$,P$_2$) package P_1 will not operate if P_2 is installed

recommends(P$_1$,P$_2$) package P_2 enhances the functionality of package P_1 in a significant way

Debian GNU/Linux System

Database

$package(\mathit{mail\text{-}reader}_1)$

$package(\mathit{mail\text{-}reader}_2)$

$package(\mathit{mail\text{-}extension})$

$package(\mathit{mail\text{-}transport\text{-}agent})$

$depends(\mathit{mail\text{-}reader}_1, \mathit{mail\text{-}transport\text{-}agent})$

$depends(\mathit{mail\text{-}reader}_2, \mathit{mail\text{-}transport\text{-}agent})$

$depends(\mathit{mail\text{-}extension}, \mathit{mail\text{-}reader}_1)$

$conflicts(\mathit{mail\text{-}reader}_1, \mathit{mail\text{-}reader}_2)$

$recommends(\mathit{mail\text{-}reader}_1, \mathit{mail\text{-}extension})$

Inference Rules

in(P) package P is chosen to be in the configuration

justified(P) package P has some reasons to be in the configuration

user-exclude(P), user-include(P) representation of user requirements

Debian GNU/Linux System

Inference Rules

$$
\begin{array}{rcl}
\{in(P)\} & \leftarrow & package(P), justified(P) \\
 & \leftarrow & in(P_1), depends(P_1, P_2), not\ in(P_2) \\
 & \leftarrow & conflicts(P_1, P_2), in(P_1), in(P_2) \\
 & \leftarrow & user\text{-}include(P), not\ in(P) \\
 & \leftarrow & user\text{-}exclude(P), in(P) \\
justified(P) & \leftarrow & user\text{-}include(P) \\
justified(P_2) & \leftarrow & depends(P_1, P_2), in(P_1) \\
justified(P_2) & \leftarrow & recommends(P_1, P_2), in(P_1)
\end{array}
$$

Suitable configurations

Given a Debian configuration model CM, a set U of user requirements, and a stable model M of $CM \cup U$, a Debian *configuration* C_M is a set of packages

$$C_M = \{P \mid in(P) \in M\}$$

Debian GNU/Linux System

Logic program

```
package(mailreader1).            depends(mailreader1,mailtransportagent).
package(mailreader2).            depends(mailreader2,mailtransportagent).
package(mailextension).          depends(mailextension,mailreader1).
package(mailtransportagent).     recommends(mailreader1,mailextension).
conflicts(mailreader1,mailreader2).

{ in(P) }     :- package(P), justified(P).
              :- in(P1), depends(P1,P2), not in(P2).
              :- conflicts(P1,P2),in(P1),in(P2).
              :- userinclude(P), not in(P).
              :- userexclude(P), in(P).
justified(P)  :- userinclude(P).
justified(P2) :- depends(P1,P2), in(P1).
justified(P2) :- recommends(P1,P2), in(P1).

userinclude(mailreader1).

hide.  show in(X).
```

Debian GNU/Linux System

Stable models

```
smodels version 2.27. Reading...done
Answer: 1
Stable Model: in(mailtransportagent) in(mailreader1)
Answer: 2
Stable Model: in(mailtransportagent) in(mailextension) in(mailreader1)
```

Debian GNU/Linux System

Logic program

```
package(mailreader1).          depends(mailreader1,mailtransportagent).
package(mailreader2).          depends(mailreader2,mailtransportagent).
package(mailextension).        depends(mailextension,mailreader1).
package(mailtransportagent).   recommends(mailreader1,mailextension).
conflicts(mailreader1,mailreader2).

{ in(P) }     :- package(P), justified(P).
              :- in(P1), depends(P1,P2), not in(P2).
              :- conflicts(P1,P2),in(P1),in(P2).
              :- userinclude(P), not in(P).
              :- userexclude(P), in(P).
justified(P)  :- userinclude(P).
justified(P2) :- depends(P1,P2), in(P1).
justified(P2) :- recommends(P1,P2), in(P1).

userinclude(mailreader2).

hide.  show in(X).
```

Debian GNU/Linux System

Stable models

```
smodels version 2.27. Reading...done
Answer: 1
Stable Model: in(mailtransportagent) in(mailreader2)
```

Debian GNU/Linux System

Logic program

```
package(mailreader1).           depends(mailreader1,mailtransportagent).
package(mailreader2).           depends(mailreader2,mailtransportagent).
package(mailextension).         depends(mailextension,mailreader1).
package(mailtransportagent).    recommends(mailreader1,mailextension).
conflicts(mailreader1,mailreader2).

{ in(P) }      :- package(P), justified(P).
               :- in(P1), depends(P1,P2), not in(P2).
               :- conflicts(P1,P2),in(P1),in(P2).
               :- userinclude(P), not in(P).
               :- userexclude(P), in(P).
justified(P)  :- userinclude(P).
justified(P2) :- depends(P1,P2), in(P1).
justified(P2) :- recommends(P1,P2), in(P1).

userinclude(mailextension).
userinclude(mailreader2).
```

No stable models!

Diagnostic Model

Unsatisfiable requirements are diagnosed using a *diagnostic model*.

The diagnostic model is constructed from the configuration model by adding a new set of atoms that represent the possible error conditions.

The diagnostic output also explains *why* each problematic component was included in the configuration.

The constraint are modified in such way that the diagnostic model will always have at least one stable model.

Diagnostic Model

New atoms

missing(P), in-conflict(P_1,P_2) denote error conditions

needs-reason(P) mark the packages that are in some way part of the problem and thus needs an explanation

user-selected(P), needs(P_1,P_2) explain why certain packages were taken into the configuration

Diagnosis

Given a Debian configuration model DM and a set U of user requirements, a *diagnosis* D is a four-tuple $D = (M, E_M, P_M, R_M)$, where

1. M is a stable model of $DM \cup U$;
2. E_M is the *error set*

$$E_M = \{missing(P) \in M\} \cup \{in\text{-}conflict(P_1, P_2) \in M\};$$

3. P_M is the *problem set*

$$P_M = \{P \mid needs\text{-}reason(P) \in M\}; \text{ and}$$

4. R_M is the *explanation set*

$$R_M = \{user\text{-}selected(P) \in M\} \cup \{needs(P_1, P_2) \in M\}.$$

Modifying configuration models

We want that the diagnosis contains only those packages that have to be in there so that it will be as small as possible.

Thus, remove rule

$$\{in(P)\} \leftarrow package(P), justified(P).$$

In addition, this ensures that no false alarms are caused by adding unnecessary recommended packages in the configuration.

Missing Packages

Error set

$missing(P)$ denote that some package in the configuration depends on P but for some reason P is not in the configuration.

Rule $\leftarrow in(P_1), depends(P_1, P_2), not\ in(P_2)$ is replaced by:

$$
\begin{aligned}
in(P_2) &\leftarrow in(P_1), depends(P_1, P_2), package(P_2) \\
missing(P_2) &\leftarrow in(P_1), depends(P_1, P_2), not\ package(P_2)
\end{aligned}
$$

The first rule ensures that existing packages are added to the model, the second rule marks non-existing packages as missing.

In addition, it may be the case that a package the user included is not available. Hence, rule $\leftarrow user\text{-}include(P), not\ in(P)$ is replaced by:

$$
\begin{aligned}
in(P) &\leftarrow user\text{-}include(P), package(P) \\
missing(P) &\leftarrow user\text{-}include(P), not\ package(P)
\end{aligned}
$$

Conflicts

Error set

$in\text{-}conflict(P_1, P_2)$ is used to model conflicts. $in\text{-}conflict(P_1, P_2)$ is true when P_1 and P_2 conflict with each other and they both have to be in the configuration.

We replace rule $\leftarrow conflicts(P_1, P_2), in(P_1), in(P_2)$ with

$$in\text{-}conflict(P_1, P_2) \leftarrow conflicts(P_1, P_2), in(P_1), in(P_2).$$

To handle the case where the user wants to leave out a package that some other package needs, we replace the rule $\leftarrow in(P), user\text{-}exclude(P)$ with

$$in\text{-}conflict(P, user\text{-}exclude) \leftarrow in(P), user\text{-}exclude(P).$$

Justifications

$needs\text{-}reason(P)$ marks the packages we want to justify.

We add the rules

$$
\begin{array}{rcl}
needs\text{-}reason(P) & \leftarrow & missing(P) \\
needs\text{-}reason(P_1) & \leftarrow & in\text{-}conflict(P_1, P_2), package(P_1) \\
needs\text{-}reason(P_2) & \leftarrow & in\text{-}conflict(P_1, P_2), package(P_2) \\
needs\text{-}reason(P_1) & \leftarrow & depends(P_1, P_2), needs\text{-}reason(P_2), in(P_1)
\end{array}
$$

$user\text{-}selected(P)$ is true when the user chose P to be in the configuration. $needs(P_1, P_2)$ is true if P_2 was included because P_1 depends on it. The justifications can be modeled with the following rules:

$$
\begin{array}{rcl}
user\text{-}selected(P) & \leftarrow & needs\text{-}reason(P), user\text{-}include(P) \\
needs(P_1, P_2) & \leftarrow & needs\text{-}reason(P_2), depends(P_1, P_2), in(P_1)
\end{array}
$$

Debian GNU / Linux System

$$
\begin{array}{rcl}
in(P_2) & \leftarrow & in(P_1), depends(P_1, P_2), package(P_2) \\
missing(P_2) & \leftarrow & in(P_1), depends(P_1, P_2), not\ package(P_2) \\
in\text{-}conflict(P_1, P_2) & \leftarrow & conflicts(P_1, P_2), in(P_1), in(P_2) \\
in\text{-}conflict(P, user\text{-}exclude) & \leftarrow & in(P), user\text{-}exclude(P) \\
in(P) & \leftarrow & user\text{-}include(P), package(P) \\
missing(P) & \leftarrow & user\text{-}include(P), not\ package(P) \\
justified(P) & \leftarrow & user\text{-}include(P) \\
justified(P_2) & \leftarrow & depends(P_1, P_2), in(P_1) \\
justified(P_2) & \leftarrow & recommends(P_1, P_2), in(P_1) \\
needs\text{-}reason(P) & \leftarrow & missing(P) \\
needs\text{-}reason(P_1) & \leftarrow & in\text{-}conflict(P_1, P_2), package(P_1) \\
needs\text{-}reason(P_2) & \leftarrow & in\text{-}conflict(P_1, P_2), package(P_2) \\
needs\text{-}reason(P_1) & \leftarrow & depends(P_1, P_2), needs\text{-}reason(P_2), in(P_1) \\
user\text{-}selected(P) & \leftarrow & needs\text{-}reason(P), user\text{-}include(P) \\
needs(P_1, P_2) & \leftarrow & needs\text{-}reason(P_2), depends(P_1, P_2), in(P_1)
\end{array}
$$

Stable Models

User requirements:

$$userinclude(mailextension). \quad userinclude(mailreader2).$$

```
package(mailreader1).           depends(mailreader1,mailtransportagent).
package(mailreader2).           depends(mailreader2,mailtransportagent).
package(mailextension).         depends(mailextension,mailreader1).
package(mailtransportagent).    recommends(mailreader1,mailextension).
conflicts(mailreader1,mailreader2).
```

Stable Models

User requirements:

$$userinclude(mailextension). \quad userinclude(mailreader2).$$

```
Stable Model: userinclude(mailreader2) userinclude(mailextension)
              needs(mailextension,mailreader1)
              in(mailreader1)
              in(mailextension)
              in(mailtransportagent)
              in(mailreader2)
              needsreason(mailreader1)
              needsreason(mailreader2)
              needsreason(mailextension)
              userselected(mailreader2)
              userselected(mailextension)
              inconflict(mailreader1,mailreader2)
```

Debian GNU / Linux System

Diagnosis

Error set:

$$E = \{in\text{-}conflict(mail\text{-}reader_1, mail\text{-}reader_2)\}$$

Problem set:

$$P = \{mail\text{-}reader_1, mail\text{-}reader_2, mail\text{-}extension\}$$

Explanation set:

$$R = \left\{ \begin{array}{l} user\text{-}selected(mail\text{-}reader_2), \\ user\text{-}selected(mail\text{-}extension), \\ needs(mail\text{-}extension, mail\text{-}reader_1) \end{array} \right\}.$$

Stable Models

User requirements:

$$userinclude(mailreader2).$$

```
Stable Model: in(mailtransportagent)
              in(mailreader2)
```

$$E = P = R = \emptyset$$

Configuration of a PC

A computer is configured using the following configuration model:

- a mass-memory $\{IDEdisk, SCSIdisk, floppydrive\}$,
- a keyboard $\{GermanlayoutKB, UKlayoutKB\}$,
- a processor $\{PII, PIII\}$,
- a motherboard $\{ATX, I820\}$,
- a graphics card $gcard$.

The following dependencies must be respected:

- SCSI disk requires an SCSI controller $\{SCSIcontroller\}$,
- $\{PII\}$ is incompatible with $I820$ and $\{PIII\}$ with ATX,
- graphics card is needed, if motherboard contains none, $\{ATX\}$ contains one, $\{I820\}$ does not.

Configuration Rule Language (CRL)

Rules are of the form

$a_1 \theta \ldots \theta a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n$

where $\theta \in \{|, \oplus\}$, $a_1, \ldots, a_n, b_1, \ldots b_m, c_1, \ldots c_n$ are atoms.

We have two types of disjunction:

$|$: "normal or"

$\oplus$: exclusive or

Configuration model of a PC

$$
\begin{array}{rcl}
computer & \leftarrow & \\
IDEdisk \mid SCSIdisk \mid floppydrive & \leftarrow & computer \\
GermanlayoutKB \oplus UKlayoutKB & \leftarrow & computer \\
PII \oplus PIII & \leftarrow & computer \\
ATX \oplus I820 & \leftarrow & computer \\
SCSIcontroller & \leftarrow & SCSIdisk \\
 & \leftarrow & PII, I820 \\
 & \leftarrow & PIII, ATX \\
gcard & \leftarrow & not\ gcardinmb \\
gcardinmb & \leftarrow & ATX
\end{array}
$$

Satisfaction

A configuration C satisfies a set of rules R in **CRL** $(C \models R)$ iff

1. If $a_1 \mid \ldots \mid a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n$,
   $\{b_1, \ldots, b_m\} \subseteq C$ and $\{c_1, \ldots, c_n\} \cap C = \emptyset$,
   then $\{a_1, \ldots a_l\} \cap C \neq \emptyset$.

2. If $a_1 \oplus \ldots \oplus a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n$,
   $\{b_1, \ldots, b_m\} \subseteq C$ and $\{c_1, \ldots, c_n\} \cap C = \emptyset$,
   then for exactly one $a \in \{a_1, \ldots a_l\}$, $a \in C$.

Computation of configurations

Given a configuration C and a set of rules R, we denote by R^C the set of rules

$$\begin{array}{rcl} \{a_i \leftarrow b_1, \ldots b_m & : & a_1\theta \ldots \theta a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n \in R, \\ & & \theta \in \{|, \oplus\}, a_i \in C, 1 \leq i \leq l, \\ & & \{c_1, \ldots, c_n\} \cap C = \emptyset\} \cup \\ \{a \leftarrow b_1, \ldots b_m & : & a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n \in R, \\ & & \{c_1, \ldots, c_n\} \cap C = \emptyset\}. \end{array}$$

The least model of R^C is denoted by $MM(R^C)$.

The configuration is *R-valid* iff $C = MM(R^C)$ and $C \models R$.

Configuration of a PC

$$
\begin{array}{rcl}
computer & \leftarrow & \\
IDEdisk \mid SCSIdisk \mid floppydrive & \leftarrow & computer \\
GermanlayoutKB \oplus UKlayoutKB & \leftarrow & computer \\
PII \oplus PIII & \leftarrow & computer \\
ATX \oplus I820 & \leftarrow & computer \\
SCSIcontroller & \leftarrow & SCSIdisk \\
 & \leftarrow & PII, I820 \\
 & \leftarrow & PIII, ATX \\
gcard & \leftarrow & not\ gcardinmb \\
gcardinmb & \leftarrow & ATX \\
GermanlayoutKB & \leftarrow &
\end{array}
$$

$$C_1 = \{computer, SCSIdisk, UKlayoutKB, PII, PIII, gcard\}$$

Configuration C_1 is not R-valid.

Configuration of a PC

$$
\begin{array}{rcl}
computer & \leftarrow & \\
IDEdisk \mid SCSIdisk \mid floppydrive & \leftarrow & computer \\
GermanlayoutKB \oplus UKlayoutKB & \leftarrow & computer \\
PII \oplus PIII & \leftarrow & computer \\
ATX \oplus I820 & \leftarrow & computer \\
SCSIcontroller & \leftarrow & SCSIdisk \\
 & \leftarrow & PII, I820 \\
 & \leftarrow & PIII, ATX \\
gcard & \leftarrow & not\ gcardinmb \\
gcardinmb & \leftarrow & ATX \\
GermanlayoutKB & \leftarrow &
\end{array}
$$

$$
C_2 = \left\{ \begin{array}{l} computer, IDEdisk, GermanlayoutKB, PIII, ATX, \\ SCSIcontroller, gcardinmb \end{array} \right\}
$$

Configuration C_2 is not R-valid.

Configuration of a PC

$$
\begin{array}{rcl}
computer & \leftarrow & \\
IDEdisk \mid SCSIdisk \mid floppydrive & \leftarrow & computer \\
GermanlayoutKB \oplus UKlayoutKB & \leftarrow & computer \\
PII \oplus PIII & \leftarrow & computer \\
ATX \oplus I820 & \leftarrow & computer \\
SCSIcontroller & \leftarrow & SCSIdisk \\
 & \leftarrow & PII, I820 \\
 & \leftarrow & PIII, ATX \\
gcard & \leftarrow & not\ gcardinmb \\
gcardinmb & \leftarrow & ATX \\
GermanlayoutKB & \leftarrow &
\end{array}
$$

$$
C_3 = \left\{ \begin{array}{l} computer, SCSIdisk, GermanlayboutKB, PII, ATX, \\ SCSIcontroller, gcardinmb \end{array} \right\}
$$

Configuration of a PC

$C_3 \models R$

Reduct R^{C_3}:

$$
\begin{array}{rcl}
computer & \leftarrow & \\
SCSIdisk & \leftarrow & computer \\
GermanlayoutKB & \leftarrow & computer \\
PII & \leftarrow & computer \\
ATX & \leftarrow & computer \\
SCSIcontroller & \leftarrow & SCSIdisk \\
 & \leftarrow & PII, I820 \\
 & \leftarrow & PIII, ATX \\
gcardinmb & \leftarrow & ATX \\
GermanlayoutKB & \leftarrow &
\end{array}
$$

$MM(R^{C_3}) = C_3$.

Configuration C_3 is R-valid.

Relationship to logic programming semantics

Let R be a set of rules in **CRL**. Let f, f' be atoms not appearing in R.

Include $f' \leftarrow not\ f', f$.

For each rule $a_1 \mid \ldots \mid a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n$ we include a rule

$$f \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n, \hat{a_1}, \ldots, \hat{a_l}$$

and for all $i = 1, \ldots, l$, two rules

$$a_i \leftarrow not\ \hat{a_i}, b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n \text{ and } \hat{a_i} \leftarrow not\ a_i$$

where $\hat{a_1}, \ldots, \hat{a_l}$ are new atoms.

For each rule $a_1 \oplus \ldots \oplus a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n$ we *additionally* include

$$f \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots not\ c_n, a', a''$$

where $a' = a_i$ and $a'' = a_j$ for some $i, j, 1 \leq i < j \leq l$.

Configuration of a PC

translation of the rules

$$computer \quad \leftarrow$$

stays unchanged.

Configuration of a PC

translation of the rules

$f' \leftarrow not\ f', f$

$$IDEdisk \mid SCSIdisk \mid floppydrive \leftarrow computer$$

$$\begin{array}{rcl}
f & \leftarrow & computer, IDEdisk', SCSIdisk', floppydrive' \\
IDEdisk & \leftarrow & not\ IDEdisk', computer \\
IDEdisk' & \leftarrow & not\ IDEdisk \\
SCSIdisk & \leftarrow & not\ SCSIdisk', computer \\
SCSIdisk' & \leftarrow & not\ SCSIdisk \\
floppydrive & \leftarrow & not\ floppydrive', computer \\
floppydrive' & \leftarrow & not\ floppydrive
\end{array}$$

Configuration of a PC

translation of the rules

$$GermanlayoutKB \oplus UKlayoutKB \leftarrow computer$$

$$\begin{aligned}
f &\leftarrow computer, GermanlayoutKB', UKlayoutKB' \\
GermanlayoutKB &\leftarrow not\ GermanlayoutKB', computer \\
GermanlayoutKB' &\leftarrow not\ GermanlayoutKB \\
UKlayoutKB &\leftarrow not\ UKlayoutKB', computer \\
UKlayoutKB' &\leftarrow not\ UKlayoutKB \\
f &\leftarrow computer, GermanlayoutKB, UKlayoutKB
\end{aligned}$$

Configuration of a PC

```
computer.
fp :- not fp, f.
f :- computer, nidedisk, nscsidisk, nfloppydrive.
idedisk :- not nidedisk, computer.
nidedisk :- not idedisk.
scsidisk :- not nscsidisk, computer.
nscsidisk :- not scsidisk.
floppydrive :- not nfloppydrive, computer.
nfloppydrive :- not floppydrive.
f:- computer, ngermanlayoutKB, nuklayoutKB.
germanlayoutKB :- not ngermanlayoutKB, computer.
ngermanlayoutKB :- not germanlayoutKB.
uklayoutKB :- not nuklayoutKB, computer.
nuklayoutKB :- not uklayoutKB.
f :- computer, germanlayoutKB, uklayoutKB.
f:- computer, npII, npIII.
pII :- not npII, computer.
npII :- not pII.
pIII :- not npIII, computer.
npIII :- not pIII.
f :- computer, pII, pIII.
f :- computer, natx, ni820.
atx :- not natx, computer.
natx :- not atx.
i820 :- not ni820, computer.
ni820 :- not i820.
f :- computer, atx, i820.
scsicontroller :- scsidisk.
:- pII, i820.
:- pIII, atx.
gcard :- not gcardinmb.
gcardinmb :- atx.
germanlayoutKB.

hide.
show computer,
idedisk,scsidisk,floppydrive,germanlayoutKB,uklayoutKB,pII,pIII,atx,i820,scsicontroller,gcard,gcardinmb.
```

Configuration of a PC

```
smodels version 2.27. Reading...done
Answer: 1
Stable Model: gcard i820 pIII germanlayoutKB idedisk computer
Answer: 2
Stable Model: gcard scsicontroller scsidisk i820 pIII germanlayoutKB idedisk computer
Answer: 3
Stable Model: gcard scsicontroller scsidisk i820 pIII germanlayoutKB floppydrive idedisk computer
Answer: 4
Stable Model: gcard i820 pIII germanlayoutKB floppydrive idedisk computer
Answer: 5
Stable Model: gcard i820 pIII germanlayoutKB floppydrive computer
Answer: 6
Stable Model: gcard scsicontroller scsidisk i820 pIII germanlayoutKB floppydrive computer
Answer: 7
Stable Model: gcard scsicontroller scsidisk i820 pIII germanlayoutKB computer
Answer: 8
Stable Model: gcardinmb atx scsicontroller scsidisk pII germanlayoutKB computer
Answer: 9
Stable Model: gcardinmb atx pII germanlayoutKB idedisk computer
Answer: 10
Stable Model: gcardinmb atx scsicontroller scsidisk pII germanlayoutKB idedisk computer
Answer: 11
Stable Model: gcardinmb atx scsicontroller scsidisk pII germanlayoutKB floppydrive idedisk computer
Answer: 12
Stable Model: gcardinmb atx scsicontroller scsidisk pII germanlayoutKB floppydrive computer
Answer: 13
Stable Model: gcardinmb atx pII germanlayoutKB floppydrive computer
Answer: 14
Stable Model: gcardinmb atx pII germanlayoutKB floppydrive idedisk computer
```

References

1. T. Syrjänen: A rule-based formal model for software configuration, 1999,
2. T. Soininen: An approach to knowledge representation and reasoning for product configuration tasks, 2000,
3. T. Syrjänen: Including diagnostic information in configuration models, 2000,
4. T. Soininen, I. Niemelä: Developing a declarative rule language for applications in product configuration, 1999.

Action languages

- Transition Systems
- Action language $\mathcal{A}$
- Action language $\mathcal{C}$

Action signatures

An *action signature* consists of three nonempty sets:

- a set $\mathbf{V}$ of *value names*,
- a set $\mathbf{F}$ of *fluent names*, and
- a set $\mathbf{A}$ of *action names*.

Intuitively, any "fluent" has a specific "value" in any "state of the world". An "action", if executed in some state, leads to a "resulting" state.

Transition systems

A *transition system* $\langle S, V, R \rangle$ of an action signature $\langle \mathbf{V}, \mathbf{F}, \mathbf{A} \rangle$ consists of

1. a set S,
2. a function V from $\mathbf{F} \times S$ into $\mathbf{V}$, and
3. a subset R of $S \times \mathbf{A} \times S$.

The elements of S are called *states*.

$V(P, s)$ is the value from P in s.

Transitions

A *transition* is any triple $\langle s, A, s' \rangle \in R$, where s' is a result of the execution of A in s.

A is *executable* in s if there is at least one such s'.

A is *deterministic* in s if there is at most one such s'.

An action signature $\langle \mathbf{V}, \mathbf{F}, \mathbf{A} \rangle$ is *propositional* if its value names are the truth values of classical logic: $\mathbf{V} = \{f, t\}$.

A transition system is propositional if its signature is propositional.

Transition graph

A transition system can be thought of as a labeled directed graph.

- States are the vertices,
- transitions $\langle s, A, s' \rangle \in R$ are represented as edges from s to s' labeled with A.

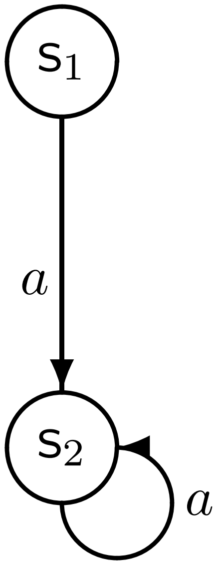
Example

Doors

$Fluents : \{closed\}$

$Actions : \{opendoor\}$

$$States : \begin{array}{rcl} s_1 & = & \{closed\} \\ s_2 & = & \{\neg closed\} \end{array}$$



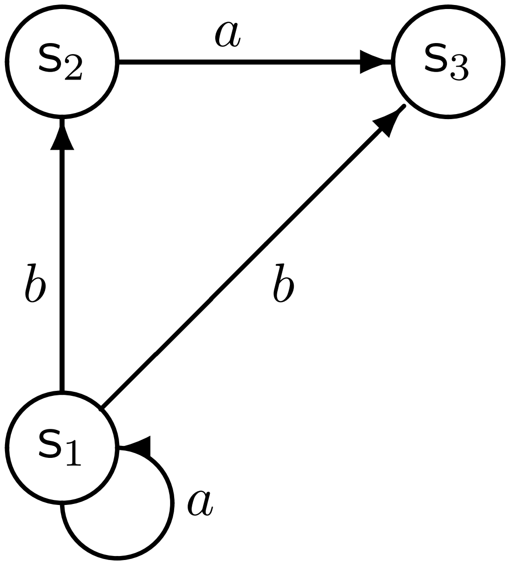
$$Transitions : \begin{array}{l} \langle s_1, a, s_2\rangle \\ \langle s_2, a, s_2\rangle \end{array}$$

Example

$Fluents : \{p, q, r\}$

$Actions : \{a, b\}$

$$
\begin{aligned}
States: \quad s_1 &= \{\neg p, \neg q, \neg r\} \\
s_2 &= \{p, q, \neg r\} \\
s_3 &= \{p, \neg q, r\}
\end{aligned}
$$



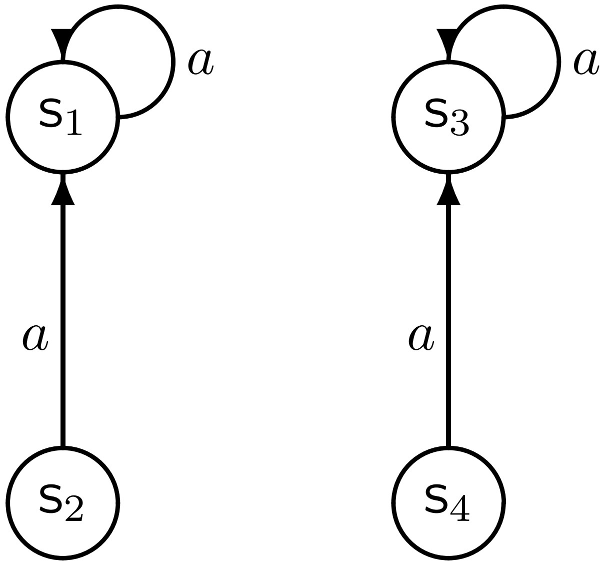
$$
\begin{aligned}
Transitions: \quad & \langle s_1, a, s_1 \rangle \\
& \langle s_1, b, s_2 \rangle \\
& \langle s_1, b, s_3 \rangle \\
& \langle s_2, a, s_3 \rangle
\end{aligned}
$$

Example

$Fluents : \{p, q\}$

$Actions : \{a\}$

$$\begin{array}{llcl} States: & s_1 & = & \{p, q\} \\ & s_2 & = & \{\neg p, q\} \\ & s_3 & = & \{p, \neg q\} \\ & s_4 & = & \{\neg p, \neg q\} \end{array}$$



$$\begin{array}{ll} Transitions: & \langle s_1, a, s_1 \rangle \\ & \langle s_2, a, s_1 \rangle \\ & \langle s_3, a, s_3 \rangle \\ & \langle s_4, a, s_3 \rangle \end{array}$$

Action language $\mathcal{A}$

Let $\langle \{f, t\}, \mathbf{F}, \mathbf{A} \rangle$ be a propositional action signature.

A *proposition* is an expression of the form

$$A \textbf{ causes } L \textbf{ if } F$$

where A is an action name, L is a literal and F is a conjunction of literals.

If F is true then **if** F can be dropped.

An *action description* is a set of propositions.

Action language $\mathcal{A}$

Let $\langle \{f, t\}, \mathbf{F}, \mathbf{A} \rangle$ be a propositional action signature. Let D be an action description in $\mathcal{A}$.

The *transition system* $\langle S, V, R \rangle$ described by D is defined as follows:

- S is the set of all interpretations of $\mathbf{F}$,
- $V(P, s) = s(P)$,
- R is set of transitions $\langle s, A, s' \rangle$ such that

$$E(A, s) \subseteq s' \subseteq E(A, s) \cup s,$$

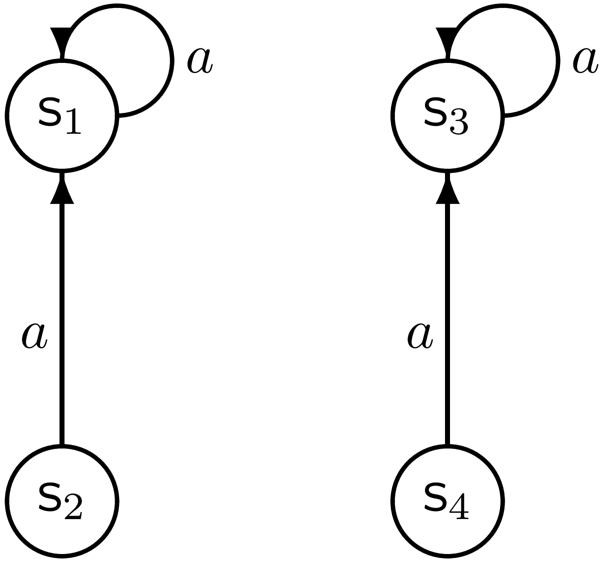
  where $E(A, s) = \{L \mid A$ **causes** L **if** F in D, s satisfies $F\}$ are the effects of A executed in s.

Example

$Fluents : \{p, q\}$ $\quad$ $Actions : \{a\}$

$Propositions : a$ **causes** p

$$
\begin{aligned}
States: \quad s_1 &= \{p, q\} \\
s_2 &= \{\neg p, q\} \\
s_3 &= \{p, \neg q\} \\
s_4 &= \{\neg p, \neg q\}
\end{aligned}
$$



$$
\begin{aligned}
Transitions: \quad & \langle s_1, a, s_1 \rangle \\
& \langle s_2, a, s_1 \rangle \\
& \langle s_3, a, s_3 \rangle \\
& \langle s_4, a, s_3 \rangle
\end{aligned}
$$

Effects of a executed in s_i are $\{p\}$ for all $i \in \{0, \ldots, 4\}$. $E(A, s) \subseteq s' \subseteq E(A, s) \cup s$ holds for all transitions $\langle s, a, s' \rangle$.

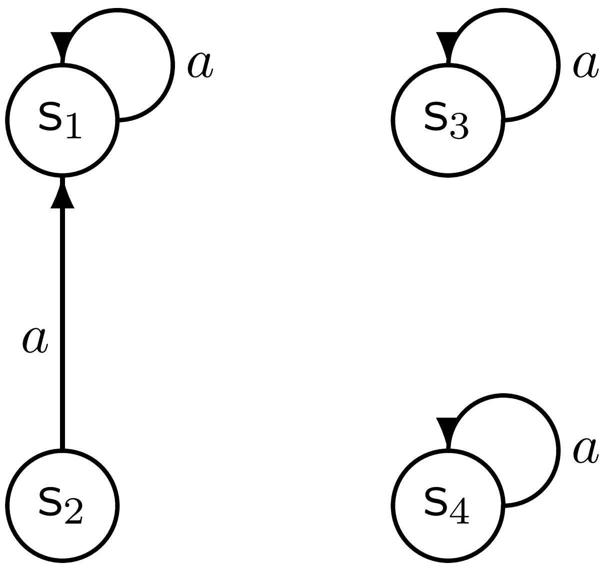
Example

$Fluents : \{p, q\}$

$Actions : \{a\}$

$Propositions : a$ **causes** p **if** q

$$
\begin{aligned}
States : \quad s_1 &= \{p, q\} \\
s_2 &= \{\neg p, q\} \\
s_3 &= \{p, \neg q\} \\
s_4 &= \{\neg p, \neg q\}
\end{aligned}
$$



$$
\begin{aligned}
Transitions : \quad & \langle s_1, a, s_1 \rangle \\
& \langle s_2, a, s_1 \rangle \\
& \langle s_3, a, s_3 \rangle \\
& \langle s_4, a, s_4 \rangle
\end{aligned}
$$

$E(a, s_1) = E(a, s_2) = \{p\}$, $E(a, s_3) = E(a, s_4) = \emptyset$. $E(A, s) \subseteq s' \subseteq E(A, s) \cup s$ holds for all transitions $\langle s, a, s' \rangle$.

Action language $\mathcal{C}$

- A *state formula* is a propositional combination of fluent names.
- A *formula* is a propositional combination of fluent names and elementary action names.
- A *static law* is an expression of the form

  caused F **if** G

  where F, G are state formulas.
- A *dynamic law* is an expression of the form

  caused F **if** G **after** U

  where F, G are state formulas and U is a formula.
- An *action description* is a set of static and dynamic laws.

Action language $\mathcal{C}$

A formula F is *caused* in a transition $\langle s, a, s' \rangle$ if it is

(i) the head of a static law

$$\textbf{caused } F \textbf{ if } G$$

such that s' satisfies G, or

(ii) the head of a dynamic law

$$\textbf{caused } F \textbf{ if } G \textbf{ after } U$$

such that s' satisfies G and $s \cup a$ satisfies U.

A transition $\langle s, a, s' \rangle$ is *causally explained* by D if there is only one s' that satisfies all formulas caused in this transition.

Action language $\mathcal{C}$

Let D be an action description in $\mathcal{C}$.

The transition system $\langle S, V, R \rangle$ described by D is as follows:

(i) S is the set of all interpretations s of $\mathbf{F}$ such that for every

caused F **if** G

in D, s satisfies F if s satisfies G,

(ii) $V(P, s) = s(P)$,

(iii) R is the set of transitions $\langle s, A, s' \rangle$ which are causally explained in D.

Example

Doors

Action: $\{OpenDoor\}$ open a springloaded door

Fluents: $\{Closed\}$

Action description:

$$
\begin{aligned}
&\textbf{caused } Closed \textbf{ if } Closed, \\
&\textbf{caused } \neg Closed \textbf{ if } True \textbf{ after } OpenDoor
\end{aligned}
$$

States: $\{\neg Closed, Closed\}$

Transitions (all causally explained):

$$
\begin{aligned}
&\langle \neg Closed, \neg OpenDoor, Closed\rangle, \\
&\langle Closed, \neg OpenDoor, Closed\rangle, \\
&\langle \neg Closed, OpenDoor, \neg Closed\rangle, \\
&\langle Closed, OpenDoor, \neg Closed\rangle.
\end{aligned}
$$

Shortcuts

A **causes** F **if** G	stands for	**caused** F **if** $True$ **after** $G \wedge A$,
inertial F	stands for	**caused** F **if** F **after** F,
inertial $F_1, \ldots F_n$	stands for	**inertial** $F_i (1 \leq i \leq n)$,
always F	stands for	**caused** $False$ **if** $\neg F$,
nonexecutable A **if** F	stands for	**caused** $False$ **after** $F \wedge A$,
default F **if** G	stands for	**caused** F **if** $F \wedge G$
exogenous F	stands for	**caused** F **if** F.

Translation to extended logic programs

Consider timesteps $t = 0, \ldots, T$

(i) **caused** F **if** $L_1 \wedge \ldots \wedge L_m$ is translated to
$F(t) \leftarrow not\ \neg L_1(t), \ldots, not\ \neg L_m(t)$,

(ii) **caused** F **if** $L_1 \wedge \ldots \wedge L_m$ **after** $L_{m+1}, \ldots, L_n$ is translated to
$F(t+1) \leftarrow not\ \neg L_1(t+1), \ldots, not\ \neg L_m(t+1), L_{m+1}(t), \ldots, L_n(t)$,

(iii) for all fluents B at time step 0 or for action names:

$$\neg B \leftarrow not\ B$$

$$B \leftarrow not\ \neg B.$$

Example

Doors

$$\textbf{default } Closed,$$

$$OpenDoor \textbf{ causes } \neg Closed$$

stands for

$$\textbf{caused } Closed \textbf{ if } Closed,$$

$$\textbf{caused } \neg Closed \textbf{ if } True \textbf{ after } OpenDoor$$

$$\begin{array}{lrcl}
(i) & Closed(t) & \leftarrow & not\ \neg Closed(t) \\
(ii) & \neg Closed(t+1) & \leftarrow & OpenDoor(t) \\
(iii) & Closed(0) & \leftarrow & not\ \neg Closed(0) \\
 & \neg Closed(0) & \leftarrow & not\ Closed(0) \\
(iii) & OpenDoor(t) & \leftarrow & not\ \neg OpenDoor(t) \\
 & \neg OpenDoor(t) & \leftarrow & not\ OpenDoor(t)
\end{array}$$

Example

Doors

```
 time(0..1).

 closed(T)  :-  not -closed(T),time(T).
-closed(T1) :-  openDoor(T),time(T),T1 = T+1,time(T1).
 closed(0)  :-  not -closed(0).
-closed(0)  :-  not  closed(0).
 openDoor(T):-  not -openDoor(T),time(T).
-openDoor(T):-  not  openDoor(T),time(T).
```

Example

Doors

```
Answer: 1
Stable Model:  closed(0)  openDoor(0) -closed(1)
Answer: 2
Stable Model: -closed(0)  openDoor(0) -closed(1)
Answer: 3
Stable Model: -closed(0) -openDoor(0)  closed(1)
Answer: 4
Stable Model:  closed(0) -openDoor(0)  closed(1)
```

Example

Monkey& Bananas

A monkey wants a bunch of bananas, hanging from the ceiling. To get the bananas the monkey must push a box to the empty place under the bananas and then climb on top of the box.

Fluents: for $x \in \{Monkey, Bananas, Box\}$

$$Loc(x),$$

$$HasBananas, OnBox$$

Actions: for $l \in \{L_1, L_2, L_3\}$

$$Walk(l), PushBox(l), ClimbOn, ClimbOff, GraspBananas$$

Example

Monkey& Bananas

caused $Loc(Bananas) = l$ **if** $HasBananas \wedge Loc(Monkey) = l$

caused $Loc(Monkey) = l$ **if** $OnBox \wedge Loc(Box) = l$

$Walk(l)$ **causes** $Loc(Monkey) = l$

nonexecutable $Walk(l)$ **if** $Loc(Monkey) = l$

nonexecutable $Walk(l)$ **if** $OnBox$

$PushBox(l)$ **causes** $Loc(Box) = l$

$PushBox(l)$ **causes** $Loc(Monkey) = l$

nonexecutable $PushBox(l)$ **if** $Loc(Monkey) = l$

nonexecutable $PushBox(l)$ **if** $OnBox$

nonexecutable $PushBox(l)$ **if** $Loc(Monkey) \neq Loc(Box)$

Example

Monkey& Bananas

$ClimbOn$ **causes** $OnBox$

nonexecutable $ClimbOn$ **if** $OnBox$

nonexecutable $ClimbOn$ **if** $Loc(Monkey) \neq Loc(Box)$

$ClimbOff$ **causes** $\neg OnBox$

nonexecutable $ClimbOff$ **if** $\neg OnBox$

$GraspBananas$ **causes** $HasBananas$

nonexecutable $GraspBananas$ **if** $HasBananas$

nonexecutable $GraspBananas$ **if** $\neg OnBox$

nonexecutable $GraspBananas$ **if** $Loc(Monkey) \neq Loc(Bananas)$

Example

Monkey& Bananas

nonexecutable $Walk(l) \wedge PushBox(l)$

nonexecutable $Walk(l) \wedge ClimbOn$

nonexecutable $PushBox(l) \wedge ClimbOn$

nonexecutable $ClimbOff \wedge GraspBananas$

exogenous c % for every action constant c

inertial c % for every simple fluent constant c

Example

Monkey& Bananas

Initial situation:

- loc(bananas,l1,0), loc(monkey,l2,0), loc(box,l3,0)

Solution:

- walk(l3,0)
  - walk(l3,0) **causes** loc(monkey,l3,1)
- pushBox(l1,1)
  - pushBox(l1,1) **causes** loc(box,l1,2)
  - pushBox(l1,1) **causes** loc(monkey,l1,2)
- climbOn(2)
  - climbOn(2) **causes** onBox(3)
- graspBananas(3)
  - graspBananas(3) **causes** hasBananas(4)

Monkey& Bananas with $\mathtt{DLV}^{\mathcal{K}}$

$\mathtt{DLV}^{\mathcal{K}}$ is a planning system, which provides an implementation of action language $\mathcal{K}$ as a front-end of the DLV system.

More information and download of the Monkey & Bananas example on:

http://www.dbai.tuwien.ac.at/proj/dlv/K/

monkey.dl

```
%% Background Knowledge

object(box).
object(monkey).
object(bananas).
```

monkey.plan

```
fluents: loc(O,L) requires object(O), #int(L).
         onBox.
         hasBananas.

actions: walk(L) requires #int(L).
         pushBox(L) requires #int(L).
         climbBox.
         graspBananas.
```

monkey.plan

```
always:  caused loc(monkey,L) after walk(L).
         caused -loc(monkey,L) after walk(L1), loc(monkey,L), L<>L1.
         executable walk(L) if not onBox.
         caused loc(monkey,L) after pushBox(L).
         caused loc(box,L) after pushBox(L).
         caused -loc(monkey,L) after pushBox(L1), loc(monkey,L), L<>L1.
         caused -loc(box,L) after pushBox(L1), loc(box,L), L<>L1.
         executable pushBox(L) if loc(monkey,L1), loc(box,L1), not onBox.
         caused onBox after climbBox.
         executable climbBox if not onBox, loc(monkey,L), loc(box,L).
         caused hasBananas after graspBananas.
         executable graspBananas if onBox, loc(monkey,L), loc(bananas,L).
         inertial loc(O,L).
         inertial onBox.
         inertial hasBananas.
```

monkey.plan

```
initially: loc(monkey,2).
           loc(box,3).
           loc(bananas,1).

noConcurrency.

goal: hasBananas ? (4)
```

Solutions

```
bash-2.05b$ dlv -FP -N=4 monkeyK.plan monkeyK.dl
DLV [build BEN/May 23 2004   gcc 2.95.4 20011002 (Debian prerelease)]

STATE 0: loc(box,3), loc(monkey,2), loc(bananas,1)
ACTIONS: walk(3)
STATE 1: loc(monkey,3), loc(box,3), -loc(monkey,2), loc(bananas,1)
ACTIONS: pushBox(1)
STATE 2: loc(monkey,1), -loc(monkey,3), -loc(box,3), loc(bananas,1), loc(box,1)
ACTIONS: climbBox
STATE 3: onBox, loc(monkey,1), loc(bananas,1), loc(box,1)
ACTIONS: graspBananas
STATE 4: loc(monkey,1), loc(bananas,1), loc(box,1), onBox, hasBananas
PLAN: walk(3); pushBox(1); climbBox; graspBananas

Check whether that plan is secure (y/n)? y
The plan is secure.

Search for other plans (y/n)? y
bash-2.05b$
```

Meta-interpreting Logic Programs

A logic program can be encoded for and interpreted by a generic meta-interpreter.

A representation $F(\Pi)$ of a logic program Π is a set of facts.

These facts are combined with a generic logic program Π_{I_α} such that

$$\mathcal{AS}(\Pi) = \{\pi(A) \mid A \in \mathcal{AS}(F(\Pi) \cup \Pi_{I_\alpha})\},$$

where π is a "simple projection function".

Representation

of a logic program

We translate Π into $F(\Pi)$ as follows:

1. For every rule

$$L_0 \leftarrow L_1, \ldots L_m, not\ L_{m+1}, \ldots not\ L_n.$$

   of Π, $F(\Pi)$ contains the facts:

$$\begin{array}{lllll} rule(r). \quad head(L_0, r). & pbl(L_1, r). & \ldots & pbl(L_m, r). \\ & nbl(L_{m+1}, r). & \ldots & nbl(L_n, r). \end{array}$$

   where r is a unique rule identifier.

2. For each pair of complementary literals $L, \neg L$ in Π we add the fact $compl(L, \neg L)$.

Example

bird & penguin

$$
\begin{array}{llll}
(1) & peng. & & \\
(2) & bird. & & \\
(3) & \neg flies & \leftarrow & not\ flies, peng. \\
(4) & flies & \leftarrow & not\ \neg flies, bird.
\end{array}
$$

$$
\begin{array}{lll}
F(\Pi): & rule(r_1). & head(peng, r_1). \\
 & rule(r_2). & head(bird, r_2). \\
 & rule(r_3). & head(negflies, r_3). \\
 & & pbl(peng, r_3).\ nbl(flies, r_3). \\
 & rule(r_4). & head(flies, r_4) \\
 & & pbl(bird, r_4).\ nbl(negflies, r_4). \\
\text{and} & & compl(flies, negflies).
\end{array}
$$

Basic Meta-interpreter program

$in\_AS(X)$ is true if literal X is in an answer set of Π.

$$\begin{aligned} in\_AS(X) \leftarrow head(X,R), \quad & pos\_body\_true(R), \\ & not\ neg\_body\_false(R). \end{aligned}$$

Basic Meta-interpreter program

The positive part of the body is true, if all its literals are in the answer set.

If there are no positive literals, the positive body is trivially true.

$$\begin{array}{rcl} pos\_body\_exists(R) & \leftarrow & pbl(X,R). \\ pos\_body\_true(R) & \leftarrow & rule(R), not\ pos\_body\_exists(R). \end{array}$$

Basic Meta-interpreter program

If positive literals exists, we proceed iteratively. We use DLV's built-in total order on constants for defining successor relation on positive body literals of each rule, and to identify the first and the last literal of a positive rule body in this total order.

Auxiliary relations:

$$\begin{aligned} pbl\_inbetween(X, Y, R) &\leftarrow pbl(X, R), pbl(Y, R), pbl(Z, R), X < Z, Z < Y. \\ pbl\_notlast(X, R) &\leftarrow pbl(X, R), pbl(Y, R), X < Y. \\ pbl\_notfirst(X, R) &\leftarrow pbl(X, R), pbl(Y, R), Y < X. \end{aligned}$$

Basic Meta-interpreter program

The positive body is true up to some positive body literal (wrt the built-in order):

$$
\begin{array}{rcl}
pos\_body\_true\_upto(R, X) & \leftarrow & pbl(X, R), not\ pbl\_notfirst(X, R), in\_AS(X). \\
pos\_body\_true\_upto(R, X) & \leftarrow & pos\_body\_true\_upto(R, Y), pbl(X, R), \\
 & & in\_AS(X), Y < X, not\ pbl\_inbetween(Y, X, R). \\
pos\_body\_true(R) & \leftarrow & pos\_body\_true\_upto(R, X), \\
 & & not\ pbl\_notlast(X, R).
\end{array}
$$

Basic Meta-interpreter program

The negative part of a body is false, if one of its literals is in the answer set.

$$neg\_body\_false(R) \leftarrow nbl(X, R), in\_AS(X).$$

Each answer set is consistent:

$$\leftarrow compl(X, Y), in\_AS(X), in\_AS(Y).$$

Each answer set A of $\Pi_{I_\alpha} \cup F(\Pi)$ represents an answer set A' of Π, where $\pi(A) = A'$ and

$$\pi(A) = \{l \mid in\_AS(l) \in A\}.$$

Meta-interpreter

```
in_AS(X) :- head(X,R), pos_body_true(R), not neg_body_false(R).
pos_body_exists(R) :- pbl(X,R).
pos_body_true(R) :- rule(R), not pos_body_exists(R).
pbl_inbetween(X,Y,R) :- pbl(X,R), pbl(Y,R), pbl(Z,R), X < Z, Z < Y.
pbl_notlast(X,R) :- pbl(X,R), pbl(Y,R), X < Y.
pbl_notfirst(X,R) :- pbl(X,R), pbl(Y,R), Y < X.
pos_body_true_upto(R,X) :- pbl(X,R), not pbl_notfirst(X,R), in_AS(X).
pos_body_true_upto(R,X) :- pos_body_true_upto(R,Y), pbl(X,R),
                           in_AS(X), Y < X, not pbl_inbetween(Y,X,R).
pos_body_true(R) :- pos_body_true_upto(R,X), not pbl_notlast(X,R).
neg_body_false(R) :- nbl(X,R), in_AS(X).
:- compl(X,Y), in_AS(X), in_AS(Y).
```

Example

bird & penguin

```
rule(r1).  head(peng,r1).
rule(r2).  head(bird,r2).
rule(r3).  head(negflies,r3).
           pbl(peng,r3). nbl(flies,r3).
rule(r4).  head(flies,r4).
           pbl(bird,r4). nbl(negflies,r4).
compl(flies,negflies).
```

```
bash-2.05b$ dlv -filter=in_AS metainterpreterAS.pl
DLV [build BEN/Apr 12 2002   gcc 2.95.3 20010315 (SuSE)]
{in_AS(peng), in_AS(bird), in_AS(flies)}
{in_AS(peng), in_AS(bird), in_AS(negflies)}
```

Crossing a river

$$
\begin{array}{lcl}
canCross & \leftarrow & boat, not\ leaking \\
canCross & \leftarrow & boat, leaking, hasBucket
\end{array}
$$

We observe somebody crossing the river with a boat. How can we explain that?

Abduction

An *abduction problem* is a triple $\langle \Pi, \mathcal{H}, \mathcal{O} \rangle$, where

- Π is a logic program,
- $\mathcal{H}$ is a set of facts, referred to as *hypotheses*, and
- $\mathcal{O}$ is a set of atoms, referred to as *observations*.

A set $\Delta \subseteq \mathcal{H}$ is an *explanation* of $\mathcal{O}$ wrt Π if all answer sets of $\Pi \cup \Delta$ contain $\mathcal{O}$.

An explanation Δ_1 is *minimal* if for every other explanation Δ_2 of $\mathcal{O}$ $\Delta_2 \not\subset \Delta_1$ holds.

We call Δ a *single* explanation if $|\Delta| = 1$.

Crossing a river

Explanations

- $\Pi = \left\{ \begin{array}{lcl} canCross & \leftarrow & boat, not\ leaking \\ canCross & \leftarrow & boat, leaking, hasBucket \end{array} \right\}$
- Hypotheses: $\mathcal{H} = \{boat, leaking, hasBucket\}$.
- Observation: $canCross$
- Explanations:
  1. $\{boat\}$
  2. $\{boat, hasBucket\}$
  3. $\{boat, leaking, hasBucket\}$
- Only $boat$ is a minimal (and single) explanation.

Running DLV

- canCross.dl

```
canCross :- boat, not leaking.
canCross :- boat, leaking, hasBucket.
```

- canCross.hyp

```
boat.
leaking.
hasBucket.
```

- canCross.obs

```
canCross.
```

Running DLV

Computing all explanations:

```
bash-2.05b$ dlv -FD canCross.dl canCross.hyp canCross.obs
DLV [build BEN/May 23 2004   gcc 2.95.4 20011002 (Debian prerelease)]

Diagnosis: boat leaking hasBucket
Diagnosis: boat
Diagnosis: boat hasBucket
```

Computing single explanations:

```
bash-2.05b$ dlv -FDsingle canCross.dl canCross.hyp canCross.obs
DLV [build BEN/May 23 2004   gcc 2.95.4 20011002 (Debian prerelease)]

Diagnosis: boat
```

Further Semantics

- Completion
- Supported Models
- Fitting semantics
- Well-founded semantics

Completion

Let Π be a basic normal logic program.
Then, Π^* is a set of rules obtained from Π as follows:

- Replace each rule $A \leftarrow$ with $A \leftarrow true$.
- If an atom A is not the head of a rule in Π, then add the rule $A \leftarrow false$.

Π^{**} is obtained from Π^* as follows:

- Replace each rule $A_0 \leftarrow A_1, \ldots A_n$ with $A_0 \leftarrow (A_1 \wedge \ldots \wedge A_n)$.
- Next, replace all rules $A \leftarrow B_1, \ldots, A \leftarrow B_m$ with the same head by $A \leftarrow (B_1 \vee \ldots \vee B_m)$.

We obtain the program *completion* of Π by replacing each occurrence of $\leftarrow$ by $\equiv$.

Example

Completion

$$\Pi : \quad \begin{array}{lcl} a & \leftarrow & \\ b & \leftarrow & a \\ c & \leftarrow & a \\ c & \leftarrow & b \\ d & \leftarrow & c, e \end{array} \qquad \Pi^* : \quad \begin{array}{lcl} a & \leftarrow & true \\ b & \leftarrow & a \\ c & \leftarrow & a \\ c & \leftarrow & b \\ d & \leftarrow & c, e \\ e & \leftarrow & false \end{array}$$

$$\Pi^{**} : \quad \begin{array}{lcl} a & \leftarrow & true \\ b & \leftarrow & a \\ c & \leftarrow & (a \vee b) \\ d & \leftarrow & (c \wedge e) \\ e & \leftarrow & false \end{array} \qquad \text{Completion of } \Pi : \quad \begin{array}{lcl} a & \equiv & true \\ b & \equiv & a \\ c & \equiv & (a \vee b) \\ d & \equiv & (c \wedge e) \\ e & \equiv & false \end{array}$$

2-valued models

- truth-values: $\{true, false\}$,
- representation: $\langle T, F \rangle$, where T is the set of all $true$ atoms and F is the set of all $false$ atoms,
- $T \cap F = \emptyset$,
- $T \cup F$ is the set of all atoms.

Supported models

A *supported model* for a (basic) normal logic program Π is a (2-valued) Herbrand model in which all equivalences of the completion of Π are true.

Example

Supported model

Completion of Π:

$$
\begin{aligned}
a &\equiv true \\
b &\equiv a \\
c &\equiv (a \vee b) \\
d &\equiv (c \wedge e) \\
e &\equiv false
\end{aligned}
$$

The supported model of Π is $\langle \{a, b, c\}, \{d, e\} \rangle$.

Including negation as failure

Π:

$$\begin{aligned} q &\leftarrow \; not\ p \\ p &\leftarrow \; not\ x, not\ q \end{aligned}$$

Completion of Π:

$$\begin{aligned} q &\equiv \neg p \\ p &\equiv (\neg x \wedge \neg q) \\ x &\equiv false \end{aligned}$$

3-valued models

- truth-values: $\{true, false, \perp\}$,
- representation: $\langle T, F\rangle$, where T is the set of all $true$ atoms and F is the set of all $false$ atoms,
- $T \cap F = \emptyset$, $T \cap \perp = \emptyset$, $F \cap \perp = \emptyset$,
- $T \cup F \cup \perp$ is the set of all atoms.

For $\langle T_1, F_1\rangle$ and $\langle T_2, F_2\rangle$ we define
$\langle T_1, F_1\rangle \leq \langle T_2, F_2\rangle$ if $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$.

Fitting Operator

Let Π be a normal logic program. The mapping Φ_Π is defined as follows:

$$\Phi_\Pi\langle T, F\rangle = \langle T', F'\rangle$$

where for all atoms A we have:

(i) $A \in T'$ if there is a rule $A \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots not\ A_n$ such that $\{A_1, \ldots, A_m\} \subseteq T$ and $\{A_{m+1}, \ldots A_n\} \subseteq F$,

(ii) $A \in F'$ if for all rules $A \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots not\ A_n$ either $\{A_1, \ldots, A_m\} \cap F \neq \emptyset$ or $\{A_{m+1}, \ldots, A_n\} \cap T \neq \emptyset$,

(iii) $A \in \bot$ otherwise.

Example

Π:

$$
\begin{array}{lll}
a & \leftarrow & \\
b & \leftarrow & a \\
c & \leftarrow & not\ a \\
d & \leftarrow & b, c
\end{array}
$$

Given 3-valued model $\langle\{a\}, \emptyset\rangle$, then
$\Phi_\Pi\langle\{a\}, \emptyset\rangle = \langle\{a, b\}, \{c\}\rangle$.

Properties of Φ_Π

Let Π be a normal logic program.

- Operator Φ_Π is monotonic, that is
  $\langle T_1, F_1\rangle \leq \langle T_2, F_2\rangle$ implies $\Phi_\Pi\langle T_1, F_1\rangle \leq \Phi_\Pi\langle T_2, F_2\rangle$.

- Φ_Π has a least fixpoint.

Fitting semantics

$$
\begin{aligned}
\Phi_\Pi^0 &= \langle \emptyset, \emptyset \rangle \\
\Phi_\Pi^{i+1} &= \Phi_\Pi(\Phi_\Pi^i) \\
\Phi_\Pi^\omega &= \cup_{i<\omega}\{\Phi_\Pi^i\}
\end{aligned}
$$

The least fixpoint of Φ_Π, denoted by Φ_Π^ω, supplies the *Fitting semantics*.

Example

Fitting

$$\Pi = \left\{ \begin{array}{lllllll} r & \leftarrow & & & s & \leftarrow & \\ t & \leftarrow & r, s & & u & \leftarrow & not\ t \\ p & \leftarrow & not\ p, not\ q & & a & \leftarrow & b \\ b & \leftarrow & a & & & & \end{array} \right\}$$

$$\begin{array}{lll} \Phi^0_\Pi & = & \langle \emptyset, \emptyset \rangle \\ \Phi^1_\Pi & = & \langle \{r, s\}, \{q\} \rangle \\ \Phi^2_\Pi & = & \langle \{r, s, t\}, \{q\} \rangle \\ \Phi^3_\Pi & = & \langle \{r, s, t\}, \{q, u\} \rangle \\ \Phi^4_\Pi & = & \Phi^3_\Pi \end{array}$$

Example

Fitting

$$\Pi = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{rcl} \Phi_\Pi^0 & = & \langle \emptyset, \emptyset \rangle \\ \Phi_\Pi^1 & = & \langle \emptyset, \emptyset \rangle \end{array}$$

But, a and b never become $true$!

$\Rightarrow$: "Extension" of Fitting semantics: *well-founded semantics*

Unfounded sets

Let Π be a normal logic program.

A set of atoms $\mathcal{A}$ is an *unfounded set* (of Π wrt $\langle T, F\rangle$) if each atom $A \in \mathcal{A}$ and for each rule $A \leftarrow A_1, \ldots A_m, not\ A_{m+1}, \ldots not\ A_n$ one of the following conditions is satisfied:

1. either $\{A_1, \ldots, A_m\} \cap F \neq \emptyset$ or $\{A_{m+1}, \ldots, A_n\} \cap T \neq \emptyset$, or
2. there is an $0 \leq i \leq m$ such that $A_i \in \mathcal{A}$.

The *greatest unfounded set* (of Π wrt $\langle T, F\rangle$) is the union of all unfounded sets (of Π wrt $\langle T, F\rangle$).

Example

Unfounded sets

$$\Pi = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

Given $\langle \emptyset, \emptyset \rangle$,
the greatest unfounded set is $\{a, b\}$.

Unfounded sets

The mapping $\mathcal{U}_\Pi$ is defined as follows:

$$\mathcal{U}_\Pi\langle T, F\rangle = \langle T', F'\rangle$$

where for all atoms A we have:

(i) $A \in T'$ if $A \in T$,

(ii) $A \in F'$ if A is in the greatest unfounded set (wrt Π and $\langle T, F\rangle$),

(iii) $A \in \perp$ otherwise.

Well-founded semantics

$$
\begin{array}{rcl}
\mathcal{W}_\Pi^0 & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{W}_\Pi^{i+1} & = & \Phi_\Pi(\mathcal{W}_\Pi^i) \cup \mathcal{U}_\Pi(\mathcal{W}_\Pi^i) \\
\mathcal{W}_\Pi^\omega & = & \cup_{i<\omega} \mathcal{W}_\Pi^i
\end{array}
$$

$\mathcal{W}_\Pi^\omega$ is the well-founded model of Π.

There always exists a well-founded model of Π.

If the well-founded model is total, then it is the unique stable model.

The well-founded model is a subset of every stable model.

Example

$$\Pi = \left\{ \begin{array}{llllllllllll} r & \leftarrow & & s & \leftarrow & & t & \leftarrow & r, s & u & \leftarrow & \mathit{not}\ t \\ p & \leftarrow & \mathit{not}\ p, \mathit{not}\ q & a & \leftarrow & b & b & \leftarrow & a & & & \end{array} \right\}$$

$$\begin{array}{lclllcl}
\Phi_\Pi^0 & = & \langle\emptyset,\emptyset\rangle & & \cup\,\mathcal{U}_\Pi^0 & = & \langle\emptyset,\emptyset\rangle \\
\mathcal{W}_\Pi^0 & = & \langle\emptyset,\emptyset\rangle & & & & \\
\Phi_\Pi^1 & = & \langle\{r,s\},\{q\}\rangle & & \cup\,\mathcal{U}_\Pi^1 & = & \langle\emptyset,\{a,b\}\rangle \\
\mathcal{W}_\Pi^1 & = & \langle\{r,s\},\{q,a,b\}\rangle & & & & \\
\Phi_\Pi^2 & = & \langle\{r,s,t\},\{q,a,b\}\rangle & & \cup\,\mathcal{U}_\Pi^2 & = & \langle\{r,s\},\{q,a,b\}\rangle \\
\mathcal{W}_\Pi^2 & = & \langle\{r,s,t\},\{q,a,b\}\rangle & & & & \\
\Phi_\Pi^3 & = & \langle\{r,s,t\},\{q,a,b,u\}\rangle & & \cup\,\mathcal{U}_\Pi^3 & = & \langle\{r,s,t\},\{q,a,b,u\}\rangle \\
\mathcal{W}_\Pi^3 & = & \langle\{r,s,t\},\{q,a,b,u\}\rangle & & & & \\
\mathcal{W}_\Pi^4 & = & \mathcal{W}_\Pi^3 & & & &
\end{array}$$

$$\mathcal{W}_\Pi^\omega = \langle\{r,s,t\},\{q,u,a,b\}\rangle$$

Relation to other systems

noMoRe:

Let Γ be the *RDG* of normal logic program Π.

$\mathcal{P}^*_\Gamma((\emptyset,\emptyset))$ corresponds to $\Phi^\omega_\Pi(\langle\emptyset,\emptyset\rangle)$, and

$\mathcal{U}_\Gamma(C)$ gives us the greatest unfounded set wrt Π and the 3-valued model obtained from C.

$(\mathcal{PU})^*_\Gamma((\emptyset,\emptyset))$ gives us the well-founded model of Π.

smodels:

$expand(\emptyset,\mathcal{A})$ gives well-founded model of Π, where $\mathcal{A}$ is the set of all atoms in Π

Alternating fixpoint characterization

Let Π be a logic program and X be a set of atoms.

- Define $C_\Pi(X) = Cn(\Pi^X)$.
- The fixpoints of $C_\Pi(X)$ are the answer sets of Π.
- $C_\Pi(X)$ is anti-monotonic. Hence, $C_\Pi(C_\Pi(X)) = C^2_\Pi(X)$ is monotonic.
- Define $A_\Pi(X) = C^2_\Pi(X)$.
- $\langle lfpA_\Pi(X), Atm \setminus C_\Pi(lfpA_\Pi(X))\rangle$ is the well-founded model of Π.

Example

Alternating fixpoint characterization

$$\Pi = \left\{ \begin{array}{lllllllllllll} r & \leftarrow & & s & \leftarrow & & t & \leftarrow & r, s & u & \leftarrow & not\ t \\ p & \leftarrow & not\ p, not\ q & a & \leftarrow & b & b & \leftarrow & a & & & \end{array} \right\}$$

$X = \{r, s, t\}$

$$Cn(\Pi^{\{r,s,t\}}) = \{r, s, t, p\} \quad Cn(\Pi^{\{r,s,t,p\}}) = \{r, s, t\} \quad A_\Pi(X) = \{r, s, t\} = X$$
$$lfpA_\Pi(X) = \{r, s, t\}$$

$$C_\Pi(lfpA_\Pi(X)) = \{r, s, t, p\} \quad Atm \setminus C_\Pi(lfpA_\Pi(X)) = \{u, a, b, q\}$$

Well-founded model: $\langle \{r, s, t\}, \{q, u, a, b\} \rangle$

The Cmodels approach

For syntactically restricted (*tight*) programs answer sets and models of completion coincide.

Cmodels idea: use program completion to compute answer sets of tight programs.

- Completion
- Basic Davis-Putnam-Logemann-Loveland procedure for SAT
- Operation of Cmodels

Completion (Recapitulation)

Let Π be a normal logic program.
Then, Π^* is a set of rules obtained from Π as follows:

- Replace each rule $A \leftarrow$ with $A \leftarrow true$.
- If an atom A is not the head of a rule in Π, then add $A \leftarrow false$.

Π^{**} is obtained from Π^* as follows:

- Replace each occurence of not with $\neg$.
- Replace each rule $A_0 \leftarrow A_1, \ldots A_n$ with $A_0 \leftarrow (A_1 \wedge \ldots \wedge A_n)$.
- Replace all rules $A \leftarrow B_1, \ldots, A \leftarrow B_m$ with same head A by $A \leftarrow (B_1 \vee \ldots \vee B_m)$.

Finally, we obtain the program *completion* of Π by replacing each occurence of $\leftarrow$ by $\equiv$.

Example

$$\Pi : \quad \begin{array}{rcl} p & \leftarrow & not\ q \\ q & \leftarrow & not\ p \\ r & \leftarrow & p \\ r & \leftarrow & q \end{array} \qquad \Pi^* = \Pi$$

$$\Pi^{**} : \quad \begin{array}{rcl} p & \leftarrow & \neg q \\ q & \leftarrow & \neg p \\ r & \leftarrow & (p \vee q) \end{array} \qquad \text{Completion of } \Pi : \quad \begin{array}{rcl} p & \equiv & \neg q \\ q & \equiv & \neg p \\ r & \equiv & (p \vee q) \end{array}$$

Satisfiability

Satisfiability problem (SAT)

> Find an interpretation satisfying a given set of propositional formulas (or determine that this set is unsatisfiable).

We consider the (superficially) simpler problem of finding an interpretation satisfying a given set of clauses
(that is, a propositional formula in CNF).

Propositional Satisfiability

- A *literal* is an atom or its negation.
- For any atom A the literals A and $\neg A$ are *complementary* to each other.
- For any literal L the literal complementary to L is denoted by $\overline{L}$.
- A *clause* is a (possibly empty) disjunction of literals.
  The empty clause is denoted by $\perp$.

☞ Any formula can be transformed into an equivalent set of clauses.

Clausification

Let F be a formula.

Clausify(F)

> *eliminate all connectives from* F *except* $\neg, \wedge$ *and* $\vee$;
>
> *distribute* $\neg$ *over* $\wedge$ *and* $\vee$ *until it applies to atoms only*;
>
> *distribute* $\vee$ *over* $\wedge$ *until it applies to literals only*;
>
> *return the set of conjunctive terms of the resulting formula*;

Clausification

We have

$$\textbf{Clausify}(p \vee \neg(q \to r)) = (p \vee q) \wedge (p \vee \neg r).$$

Problem: **Clausify**(F) can be much longer than F; for instance if F is

$(p_1 \wedge q_1) \vee \ldots \vee (p_n \wedge q_n)$ *then Clausify*(F) *consists of* 2^n *clauses.*

☞ Use alternative clausification.

Clausification with new atoms

Let F be a formula and let Γ be set of clauses (initially $\Gamma = \emptyset$).

Clausify$^\star(F, \Gamma)$

if F *is a conjunction of clauses* $C_1 \wedge \ldots \wedge C_k$

then *exit with* $\{C_1, \ldots, C_k\} \cup \Gamma$;

$G \leftarrow$ *a minimal (wrt subformulas) non-literal subformula of* F;

$u \leftarrow$ *a new atom*;

$F \leftarrow$ *result of replacing* G *in* F *by* u;

Clausify$^\star(F, \Gamma \cup$ *Clausify*$(u \equiv G))$;

☞ *Clausify*$^\star(F, \emptyset)$ contains atoms not contained in F.

Example for Clausify$^\star(F, \Gamma)$

F	Γ	G	u	**Clausify**$(u \equiv G)$
$p \vee \neg(q \rightarrow r)$	$\emptyset$	$q \rightarrow r$	u_0	$(\neg u_0 \vee \neg q \vee r) \wedge (q \vee u_0) \wedge (\neg r \vee u_0)$
$p \vee \neg u_0$	$\emptyset$			

☞ *Clausify*$^\star(F, \Gamma) = (p \vee \neg u_0) \wedge (\neg u_0 \vee \neg q \vee r) \wedge (q \vee u_0) \wedge (\neg r \vee u_0)$

Properties of Clausify*

- F and *Clausify*$^\star(F, \emptyset)$ are not equivalent:

  To see this consider

$$\begin{array}{rcl} F & = & p \vee \neg(q \rightarrow r) \\ \textit{Clausify}^\star(F, \Gamma) & = & (p \vee \neg u_0) \wedge (\neg u_0 \vee \neg q \vee r) \wedge (q \vee u_0) \wedge (\neg r \vee u_0) \end{array}$$

  Let I be an interpretation s.t. $I(q) = I(u_0) = 1$ and $I(p) = I(r) = 0$.
  Then $I(F) = 1$ but $I(\textit{Clausify}^\star(F, \Gamma)) = 0$.

- Each model of F can be extended to "new" atoms such that *Clausify*$^\star(F, \emptyset)$ will be satisfied (take $I(u_0) = 0$).

- Each model of *Clausify*$^\star(F, \emptyset)$ is a model of F when restricted to "old" atoms.

Example for Clausify$^\star(F, \Gamma)$

F	Γ	G	u	**Clausify**$(u \equiv G)$
$(p_1 \wedge q_1) \vee (p_2 \wedge q_2)$	$\emptyset$	$p_1 \wedge q_1$	u_0	$(\neg u_0 \vee p_1) \wedge (\neg u_0 \vee q_1) \wedge$ $(\neg p_1 \vee \neg q_1 \vee u_0)$
$u_0 \vee (p_2 \wedge q_2)$	**Clausify**$(u_0 \equiv p_1 \wedge q_1) \cup \Gamma$	$p_2 \wedge q_2$	u_1	$(\neg u_1 \vee p_2) \wedge (\neg u_1 \vee q_2) \wedge$ $(\neg p_2 \vee \neg q_2 \vee u_1)$
$u_0 \vee u_1$	**Clausify**$(u_1 \equiv p_2 \wedge q_2) \cup \Gamma$			

$$
\begin{aligned}
\textit{Clausify}^\star(F, \Gamma) \quad = \quad & (u_0 \vee u_1) \wedge \\
& (\neg u_0 \vee p_1) \wedge (\neg u_0 \vee q_1) \wedge (\neg p_1 \vee \neg q_1 \vee u_0) \wedge \\
& (\neg u_1 \vee p_2) \wedge (\neg u_1 \vee q_2) \wedge (\neg p_2 \vee \neg q_2 \vee u_1)
\end{aligned}
$$

For $F = (p_1 \wedge q_1) \vee \ldots \vee (p_n \wedge q_n)$ we have that *Clausify*$^\star(F, \Gamma)$ has $(3n + 1)$ clauses.

☞ This is small compared to 2^n clauses of *Clausify*(F) (take $n = 5$).

Unit Clause Propagation

A *unit* clause is a clause that consists of a single literal.

If a set of clauses contains a unit clause, then it can be simplified using the fact that

- the set of clauses $\{F, F \vee G\}$ is equivalent to $\{F\}$ and
- the set of clauses $\{F, \neg F \vee G\}$ is equivalent to $\{F, G\}$.

A simplification step like this may create a new unit clause and that can make further simplifications possible.

☞ This process is called *unit propagation*.

Unit Clause Propagation Procedure

UnitPropagation(Γ, U)

while there is a unit clause $\{L\}$ in Γ

$U \leftarrow U \cup \{L\}$

for *every clause* $C \in \Gamma$ **do**

if $L \in C$ **then** $\Gamma \leftarrow \Gamma \setminus \{C\}$

elsif $\overline{L} \in C$ **then** $\Gamma \leftarrow (\Gamma \setminus \{C\}) \cup \{C \setminus \{\overline{L}\}\}$

end

end

- Γ is a set of clauses;
- U is a consistent set of literals such that, for every $L \in U$, neither L nor $\overline{L}$ occurs in any clause in Γ.

Some remarks

- During execution, Γ is simplified and U grows bigger.
- Upon termination, there are no unit clauses in Γ.
- To apply unit clause propagation to a set of clauses Γ_0, the procedure is invoked with $\Gamma = \Gamma_0$ and $U = \emptyset$.
- After every execution of the while-loop, $\Gamma \cup U$ remains equivalent to the original set of clauses Γ_0.

Example

For $\Gamma = \{p, \neg p \vee \neg q, \neg q \vee r\}$ apply *Unit-Propagation*.

Γ	U	L
$\{p, \neg p \vee \neg q, \neg q \vee r\}$	$\emptyset$	p
$\{\neg q, \neg q \vee r\}$	$\{p\}$	$\neg q$
$\emptyset$	$\{p, \neg q\}$	*no*

☞ This computation shows that $\Gamma = \{p, \neg p \vee \neg q, \neg q \vee r\}$ is equivalent to $\{p, \neg q\}$.

☞ Each interpretation I with $I(p) = 1$ and $I(q) = 0$ is a model of Γ.

Another Example

For $\Gamma = \{p, p \vee q, \neg p \vee \neg q, q \vee r, \neg q \vee \neg r\}$ apply *Unit-Propagation*.

Γ	U	L
$\{p, p \vee q, \neg p \vee \neg q, q \vee r, \neg q \vee \neg r\}$	$\emptyset$	p
$\{\neg q, q \vee r, \neg q \vee \neg r\}$	$\{p\}$	$\neg q$
$\{r\}$	$\{p, \neg q\}$	r
$\emptyset$	$\{p, \neg q, r\}$	*no*

☞ Interpretation I with $I(p) = I(r) = 1$ and $I(q) = 0$ is a model of Γ.

Properties of Unit-Propagation

There are two cases when unit clause propagation alone is sufficient for solving SAT for a set of clauses Γ_0.

For this, consider the values of Γ and U upon termination of *Unit-Propagation*$(\Gamma_0, \emptyset)$.

1. if Γ includes the empty clause, then Γ is unsatisfiable, and so is Γ_0;
2. if $\Gamma = \emptyset$, then Γ_0 is equivalent to U, which is a consistent set of literals.

   A model of Γ_0 can easily obtained from U.

Davis-Putnam-Logemann-Loveland Procedure

The DPLL procedure is an extension to unit clause propagation that allows for solving SAT in full generality.

Observation

For any set of formulas Γ and any formula F,

the set of models of Γ is the union of the sets of models of $\Gamma \cup \{F\}$ and $\Gamma \cup \{\neg F\}$.

The DPLL procedure uses this fact to apply unit clause propagation (even) when Γ does not contain unit clauses.

The DPLL algorithm

DPLL(Γ, U**)**

> **UnitPropagation(**Γ, U**)**
>
> **if** $\emptyset \in \Gamma$ **then return**
>
> **if** $\Gamma = \emptyset$ **then exit with** a model of U
>
> $A \leftarrow$ **select(**$atoms(\Gamma)$**)**
>
> **DPLL(**$\Gamma \cup \{A\}, U$**)**
>
> **DPLL(**$\Gamma \cup \{\overline{A}\}, U$**)**

- Γ is a set of clauses; U is a consistent set of literals such that, for every $L \in U$, neither L nor $\overline{L}$ occurs in any clause in Γ (see above).
- $atoms(\Gamma)$ returns the set of all atoms occuring in Γ,
- Initially, **DPLL** is invoked with $\Gamma = \Gamma_0$ and $U = \emptyset$.

Some remarks

- The *return* of the first if-statement indicates that $\Gamma \cup U$ is unsatisfiable.
- The *exit* in the second statement produces a model of $\Gamma \cup U$.

Example

Let $\Gamma = \{\neg p \vee q, \neg p \vee r, q \vee r, \neg q \vee \neg r\}$ be a set of clauses and apply *DPLL*.

$\Gamma = \{\neg p \vee q, \neg p \vee r, q \vee r, \neg q \vee \neg r\}$

$U = \emptyset$

DPLL(Γ, U)

UP leaves Γ and U unchanged

DPLL$(\Gamma \cup \{p\}, U)$

UP gives $\emptyset \in \Gamma$ hence no model

DPLL$(\Gamma \cup \{\neg p\}, U)$

UP gives

$\Gamma = \{q \vee r, \neg q \vee \neg r\}$

$U = \{\neg p\}$

select q

DPLL$(\Gamma \cup \{q\}, U)$

UP gives

$\Gamma = \emptyset$

$U = \{\neg p, q, \neg r\}$ is model

DPLL$(\Gamma \cup \{\neg q\}, U)$

UP gives

$\Gamma = \emptyset$

$U = \{\neg p, \neg q, r\}$ is model

Operation of Cmodels

In the process of its operations, Cmodels

1. simplifies the given normal program,
2. verifies that the resulting program is tight,
3. forms the program completion and call a SAT solver to find its models.

Simplification

Let $AS(\Pi)$ denote the set of all answer sets of program Π.

- Two programs Π and Π' are *equivalent* if $AS(\Pi) = AS(\Pi')$.

☞ Many other different equivalence concepts were proposed in the literature!

Some Definitions

Define

- $Atom(\Pi)$ — all atoms occurring in Π
- $Atom^+(\Pi) = \bigcap_{X \in AS(\Pi)} X$ — intersection of all answer sets
- $Atom^-(\Pi) = Atom(\Pi) \setminus \bigcup_{X \in AS(\Pi)} X$ — atoms not in any answer set of Π

Simplification

Any program Π of the form

$$\begin{array}{l} \textit{Head} \leftarrow \textit{Body}, F. \\ \Pi' \end{array}$$

s.t. $F \in Atom^{-}(\Pi)$ is equivalent to program

$$\begin{array}{l} \leftarrow F. \\ \Pi' \end{array} \tag{2}$$

Simplification

Let Π be a program of the form

$$\begin{array}{l} \textit{Head} \leftarrow \textit{Body}, \ not\ F. \\ \Pi' \end{array}$$

- If $F \in Atom^+(\Pi)$ then Π is equivalent to program

$$\begin{array}{l} \leftarrow not\ F. \\ \Pi' \end{array} \tag{3}$$

- If $F \in Atom^-(\Pi)$ then Π is equivalent to program

$$\begin{array}{l} \leftarrow F. \\ \textit{Head} \leftarrow \textit{Body}. \\ \Pi' \end{array} \tag{4}$$

Example

$$\Pi = \begin{array}{l} a \leftarrow b, not\ c. \\ d \leftarrow not\ a. \\ c \leftarrow not\ d. \end{array}$$

☞ $AS(\Pi) = \{\{d\}\}$ implies $Atom^+(\Pi) = \{d\}$ and $Atom^-(\Pi) = \{a, b, c\}$.

According to (2) Π is equivalent to

$$\Pi' = \begin{array}{l} \leftarrow b. \\ d \leftarrow not\ a. \\ c \leftarrow not\ d. \end{array}$$

☞ $Atom^+(\Pi') = Atom^+(\Pi) = \{d\}$ and $Atom^-(\Pi') = Atom^-(\Pi) = \{a, b, c\}$.

Example ctd

According to (3) Π' is equivalent to

$$\Pi'' = \begin{array}{l} \leftarrow b. \\ d \leftarrow not\ a. \\ \leftarrow not\ d. \end{array}$$

According to (4) Π'' is equivalent to

$$\Pi''' = \begin{array}{l} \leftarrow b. \\ \leftarrow a. \\ d \leftarrow . \\ \leftarrow not\ d. \end{array}$$

Remark on Simplification

How to determine sets $Atom^+(\Pi)$ and $Atom^-(\Pi)$?

- for example, all facts of Π are in $Atom^+(\Pi)$
- for example, atoms $\{a \mid (\leftarrow a) \in \Pi\}$ are in $Atom^-(\Pi)$
- cmodels uses smodels procedures *atmost* and *atleast* to determine subsets of $Atom^+(\Pi)$ and $Atom^-(\Pi)$, respectively.
- In general, the WFM is part of any answer set.

Tight Programs

The *positive dependency graph* G of a normal logic program Π is a directed graph s.t.

1. the vertexes of G are the atoms occurring in Π
2. G has an edge from A to B if Π has a rule with head B that contains A in the positive part of the body.

A program is *tight* if its positive dependency graph has no cycles.

Example

$$\Pi = \begin{array}{l} p \leftarrow not\ p. \\ p \leftarrow q. \\ q \leftarrow p. \end{array}$$

The positive dependency graph of Π:

☞ The graph has a cycle and thus Π is not tight.

Tightness

Before verifying tightness cmodels applies some further simplifications

- rules of the form $A \leftarrow \ldots, A, \ldots$ are deleted since this does not change answer sets and
- "inessential" vertexes are eliminated, for example atoms that do not occur in the head of a rule.

Then cmodels verifies tightness by

1. building the positive dependency graph and
2. uses a standard depth first search algorithm to detect cycles.

Example

Let Π be the following program:

$$a \leftarrow not\ b, not\ d.$$

$$b \leftarrow not\ a, not\ c.$$

$$c \leftarrow d.$$

$$d \leftarrow c.$$

$$f \leftarrow c, not\ f.$$

$$c \leftarrow not\ d, not\ a, not\ b.$$

$$d \leftarrow not\ c, not\ a, not\ b.$$

☞ Which answer sets?

Example

Π	$Sim(\Pi)$	*Completion*
$a \leftarrow not\ b, not\ d.$	$a \leftarrow not\ b.$	$a \equiv \neg b$
$b \leftarrow not\ a, not\ c.$	$b \leftarrow not\ a.$	$b \equiv \neg a$
$c \leftarrow d.$	$\leftarrow d.$	$\bot \equiv d.$
$d \leftarrow c.$	$\leftarrow c.$	$\bot \equiv c.$
$f \leftarrow c, not\ f.$	$\leftarrow c.$	
$c \leftarrow not\ d, not\ a, not\ b.$	$c \leftarrow not\ a, not\ b.$	$c \equiv \neg b \wedge \neg a.$
$d \leftarrow not\ c, not\ a, not\ b.$	$d \leftarrow not\ a, not\ b.$	$d \equiv \neg b \wedge \neg a.$

☞ Π is not tight, but $Sim(\Pi)$ is tight.

☞ $AS(\Pi) = \{\{a\}, \{b\}\}$.

Detailed Operation of Cmodels

In the process of its operations, Cmodels

1. simplifies a given program generated from **lparse**
2. turns it into a **basic** nested program
3. verifies that the resulting program is tight,
4. forms the program completion and call a SAT solver to find its models.

Iparse Programs

An output program of lparse contains

- normal rules of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n,$$

- choice rules of the form

$$\{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_k,$$

- Weight constraint rules of the form

$$A_0 \leftarrow l\ \{A_1 = w_1, \ldots, A_m = w_m\},$$

where each A_i is an atom, and l (lower bound) and w_j (weights) are integers.

Basic Nested Programs

A *basic nested program* contains

- basic nested rules of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n, not\ not\ A_{n+1}, \ldots, not\ not\ A_k,$$

where each A_i is an atom.

☞ This is a special case of the concept of programs with nested expressions.

☞ Observe that the definiton of tightness works for basic nested without changes.

Example: choice rules

The following lparse program

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

is translated by cmodels into the following program with nested expressions:

$$
\begin{array}{lll}
p & \leftarrow & not\ not\ p \\
q & \leftarrow & not\ not\ q \\
r & \leftarrow & p \\
r & \leftarrow & q
\end{array}
$$

Simplification

Any lparse program of the form

$$\begin{array}{lcl} \textit{Head} & \leftarrow & L\{F = w, Tail\} \\ \Pi' & & \end{array}$$

s.t. $F \in Atom^{-}(\Pi)$ is equivalent to program

$$\begin{array}{lcl} & \leftarrow & F \\ \textit{Head} & \leftarrow & L\{Tail\} \\ \Pi' & & \end{array} \tag{5}$$

Simplification

Let Π be a lparse program of the form

$$\textit{Head} \leftarrow L\{not\ F = w, Tail\}$$
$$\Pi'$$

- If $F \in Atom^+(\Pi)$ then Π is equivalent to program

$$\begin{array}{l} \leftarrow not\ F. \\ \textit{Head} \leftarrow L\{Tail\} \\ \Pi' \end{array} \tag{6}$$

- If $F \in Atom^-(\Pi)$ then Π is equivalent to program

$$\begin{array}{l} \leftarrow F. \\ \textit{Head} \leftarrow L - w\{Tail\} \\ \Pi' \end{array} \tag{7}$$

Example

Let Π be the following program:

$$\{a\}$$

$$c \leftarrow 4\{a = 1, b = 1, not\ d = 2\}$$

☞ $AS(\Pi) = \{\{a\}, \emptyset\}$ implies $Atom^+(\Pi) = \emptyset$ and $Atom^-(\Pi) = \{b, c, d\}$.

According to (5) Π is equivalent to

$$\{a\}$$

$$\leftarrow b$$

$$c \leftarrow 4\{a = 1, not\ d = 2\}$$

Example ctd

Since $Atom^{-}(\Pi) = \{b, c, d\}$, according to (7) program

$$\{a\}$$

$$\leftarrow b$$

$$c \leftarrow 4\{a = 1, not\ d = 2\}$$

is equivalent to

$$\{a\}$$

$$\leftarrow b$$

$$\leftarrow d$$

$$c \leftarrow 2\{a = 1\}.$$

Example: Translation to Basic Nested Programs

The rule

$$p \leftarrow 3\{q = 3, r = 2, s = 2\}$$

is translated to four simpler rules:

$$p \leftarrow \textit{aux1}$$

$$\textit{aux1} \leftarrow 3\{r = 2, s = 2\}$$

$$p \leftarrow q, \textit{aux2}$$

$$\textit{aux2} \leftarrow 0\{r = 2, s = 2\}$$

☞ *aux1* is an abbreviation for $3\{r = 2, s = 2\}$ and

☞ *aux2* is an abbreviation for $0\{r = 2, s = 2\}$.

☞ since $0\{r = 2, s = 2\}$ is always true *aux2* can be dropped.

Example ctd

We obtain

$$p \leftarrow \textit{aux1}$$

$$\textit{aux1} \leftarrow 3\{r = 2, s = 2\}$$

$$p \leftarrow q$$

A similar step gives:

$$p \leftarrow \textit{aux1}$$

$$\textit{aux1} \leftarrow \textit{aux3}$$

$$\textit{aux3} \leftarrow 3\{s = 2\}$$

$$\textit{aux1} \leftarrow r, \textit{aux4}$$

$$\textit{aux4} \leftarrow 1\{s = 2\}$$

$$p \leftarrow q$$

☞ *aux3* is an abbreviation for $3\{s = 2\}$ and

☞ *aux4* is an abbreviation for $1\{s = 2\}$.

Example ctd

Since $3\{s = 2\}$ is always false we obtain program

$$p \leftarrow \mathit{aux1}$$
$$\mathit{aux1} \leftarrow r, \mathit{aux4}$$
$$\mathit{aux4} \leftarrow 1\{s = 2\}$$
$$p \leftarrow q$$

Finally, we get the following program since $1\{s = 2\}$ can be replaced by s:

$$p \leftarrow \mathit{aux1}$$
$$\mathit{aux1} \leftarrow r, \mathit{aux4}$$
$$\mathit{aux4} \leftarrow s$$
$$p \leftarrow q$$

SAT for Horn Clauses and Computation of the least Herbrand Modell

Clauses vs Rules:
The efficient DPLL algorithm for SAT can be further optimized if we restrict ourself to clauses with “direction”.

☞ We deal with definite programs, Horn clauses or rules.

- efficient procedure for SAT on Horn clauses
- efficient computation of the least Herbrand model
- naive algorithm
- optimized algorithm according to Dowling und Gallier (1984)
- application to ASP solvers

Horn-Klauseln (Erinnerung)

Eine aussagenlogische Horn-Klausel ist

- eine definite Programmklausel oder
- ein definites Ziel

☞ eine Horn-Klausel hat folgende Form:

- A (Fakt)
- $[A, \neg B_1, \ldots, \neg B_m]$ ($A \leftarrow B_1 \wedge \ldots \wedge B_m$) (definite Klausel kein Fakt)
- $[\neg B_1, \ldots, \neg B_m]$ ($\leftarrow B_1 \wedge \ldots \wedge B_m$) (definites Ziel)

☞ Horn-Klauseln und definite Programme unterscheiden sich nur durch definite Ziele

Beispiel

Betrachte folgende Menge von Horn-Klauseln:

$$A$$

$$\neg A \vee B$$

$$\neg A \vee \neg B$$

☞ Horn-Klauseln können inkonsistent sein, aber

☞ definite Programme (als Klauselmenge) sind immer konsistent.

Das Kleinste Herbrand-Modell (Erinnerung)

Sei $\mathcal{P}$ ein definites aussagenlogisches Programm und I ein Interpretation.

$$T_{\mathcal{P}}(I) = \{A \mid (A \leftarrow B_1 \wedge \ldots \wedge B_m) \in \mathcal{P} \text{ und } \{B_1, \ldots, B_n\} \subseteq I\}$$

$$T_{\mathcal{P}} \uparrow 0 = \emptyset$$

$$T_{\mathcal{P}} \uparrow (n+1) = T_{\mathcal{P}}(T_{\mathcal{P}} \uparrow n) \quad (\text{für } n \in \mathbb{N})$$

☞

$$T_{\mathcal{P}} \uparrow 1 = \{A \mid (A \leftarrow B_1 \wedge \ldots \wedge B_m) \in \mathcal{P} \text{ und } \{B_1, \ldots, B_n\} \subseteq \emptyset\}$$

$$T_{\mathcal{P}} \uparrow (n+1) = \{A \mid (A \leftarrow B_1 \wedge \ldots \wedge B_m) \in \mathcal{P} \text{ und } \{B_1, \ldots, B_n\} \subseteq T_{\mathcal{P}} \uparrow (n)\}$$

Algorithmus 1 für definite Programme

Sei $\mathcal{P} = \{r_1, \ldots, r_n\}$ ein definites Programm, $Atm(\mathcal{P}) = \{A_1, \ldots, A_m\}$ die aussagenlogischen Variablen von $\mathcal{P}$, $\vec{V}$ ein boolscher Vektor der Länge m und ch eine boolsche Variable.

program algorithm1 ;
begin $ch := t$;
for each $X \in Atm(\mathcal{P})$ **do** $V(X) := f$ **end for**
for each $X \in Atm(\mathcal{P})$ s.t. $X \in \mathcal{P}$ **do** $V(X) := t$ **end for**
  while ch **do**
    $ch := f;$
    for each $(A \leftarrow B_1 \wedge \ldots \wedge B_k) \in \mathcal{P}$ **do**
      if $V(B_1) = t, \ldots, V(B_k) = t$ **and** $V(A) = f$ **then**
        $V(A) := t;\ ch := t;$
        $\mathcal{P} := \mathcal{P} \setminus \{A \leftarrow B_1 \wedge \ldots \wedge B_k\}$
      end if
    end for
  end while
end

☞ $M_{\mathcal{P}} = \{A \mid V(a) = t\}$ mit Komplexität $O(n^2)$.

Beispiel

Betrachte folgendes definite Programm (Ordnung nicht relevant):

$$
\begin{array}{ll}
A & B \leftarrow A \\
C & D \leftarrow B \wedge C \\
E \leftarrow D \wedge F & F \leftarrow E
\end{array}
$$

- $V(A) = V(B) = V(C) = V(D) = V(E) = V(F) = V(G) = f$;
- $V(A) = V(C) = t$
- 1. While-Loop: $V(B) = t$; $\mathcal{P} := \mathcal{P} \setminus \{B \leftarrow A\}$
- 2. While-Loop: $V(D) = t$; $\mathcal{P} := \mathcal{P} \setminus \{D \leftarrow B \wedge C\}$
- 3. While-Loop: keine weitern Änderungen

Algorithmus 1 für Horn-Klauseln

Sei $\mathcal{H} = \{r_1, \ldots, r_n\}$ ein definites Programm, $Atm(\mathcal{H}) = \{A_1, \ldots, A_m\}$ die aussagenlogischen Variablen, $\vec{V}$ ein boolscher Vektor der Länge m und ch und $cons$ boolsche Variablen.

program algorithm1;
begin $ch := t$; $cons := t$;
for each $X \in Atm(\mathcal{H})$ **do** $V(X) := f$ **end for**
for each $X \in Atm(\mathcal{H})$ s.t. $X \in \mathcal{H}$ **do** $V(X) := t$ **end for**
  while ch **and** $cons$ **do** $ch := f$;
    for each $C \in \mathcal{H}$ **and** ch **do**
      if $C = \neg B_1 \vee \ldots \vee \neg B_k$ **and** $V(B_1) = t, \ldots, V(B_k) = t$ **then** $cons := f$
      else
        if $C = (A \vee \neg B_1 \vee \ldots \vee \neg B_k)$ **and** $V(B_1) = t, \ldots, V(B_k) = t$ **and** $V(A) = f$ **then**
          $V(A) := t$; $ch := t$;
          $\mathcal{H} := \mathcal{H} \setminus C$
        end if
      end if
    end for
  end while end

Beispiel

Betrachte folgende Menge von Horn-Klauseln (Ordnung nicht relevant):

$$A$$

$$\neg A \vee B$$

$$\neg A \vee \neg B$$

- ➥ $V(A) = V(B) = f$;
- ➥ $V(A) = t$
- ➥ 1. While-Loop: $V(B) = t$; $\mathcal{H} := \mathcal{H} \setminus \{\neg A \vee B\}$
- ➥ 2. While-Loop: $cons = f$;

☞ Inkonsistenz entdeckt!!

☞ Komplexität $O(n^2)$

Optimierung von Algorithmus 1

☞ Initial werden alle aussagenlogischen Variablen gleich f (false) gesetzt (wie bisher), **aber** es wird andere Datenstruktur verwendet.

- für jede Horn-Klausel $C \in \mathcal{H}$ wird ein Zähler eingeführt, dessen Wert die Anzahl der negativen Literale mit Wahrheitswert f (false) angibt (falls es keine negativen Literale in c gibt ist der Wert 0)
- eine Klausel C wird bearbeitet, wenn der Wert ihres Zählers 0 ist, d.h. für alle negativen Literale $\neg B \in C$ ist $B = t$ (true)
  ➥ das positive Literal $A \in C$ wird auf t (true) gesetzt sobald der Zähler von C null ist
- mit jeder aussagenlogischen Variabel B ist die Liste der Klauseln assoziiert, in denen B negative vorkommt
- ist ein positives Literal $A = t$ (true), werden die Zähler aller Klauseln in denen A negativ vorkommt um eins decrementiert

Algorithmus 2 für Horn-Klauseln

Sei $\mathcal{H} = \{r_1, \ldots, r_n\}$ ein definites Programm und
$Atm(\mathcal{H}) = \{A_1, \ldots, A_m\}$ die aussagenlogischen Variablen, die in $\mathcal{H}$ vorkommen.

program algorithm2;

type $clause$ = **record** $N : 1 \,..\, n$; $next$: ^ $clause$ **end** ;

type lit = **record** val : **boolean** ; $clauselist$: ^ $clause$ **end** ;

type $Hornclause$ = **array** [$1 \,..\, m$] **of** lit ;

type $count$ = **array** [$1 \,..\, n$] **of** $0 \,..\, m$;

var $\mathcal{H}$: $Hornclause$; $queue$: q-type ;

$num, poslit$: $count$; $cons$: **boolean** ;

begin

input($\mathcal{H}$);

init($\mathcal{H}, num, poslit, queue, new$);

$cons := t$;

sat($\mathcal{H}, num, poslit, queue, cons, new$);

if $cons$ **then**

print($assignment$);

else

print($unsat$);

end

Algorithmus 2 Initialisierung

Gegeben: $\mathcal{H}$

Initialisierung von num, $poslit$ und $cons$ ($queue$):

1. für alle Variablen $A \in Atm(\mathcal{H})$ setzte
   - $A.val := f$ und
   - $A.clauselist :=$ Liste aller $C \in \mathcal{H}$ mit $\neg A \in C$
2. für alle Klauseln $C \in \mathcal{H}$ setzte
   - $num[C] :=$ Anzahl der negativen Literale in C und
   - $poslit[C] := \begin{cases} A & \text{falls } A \text{ positive in } C \\ 0 & \text{sonst (steht für false)} \end{cases}$
3. setzte $queue :=$ Liste aller Fakten,
   d.h. aller Klauseln ohne negative Literale
4. setzte $cons := t$

Algorithmus 2 sat

```
program  sat(H, num, poslit, queue, cons, new);
var  c1, c2, old, new, next : 1 .. n;            \* H = {r_1, ..., r_n} *\
     k : 1 .. m;                                  \* Atm(H) = {A_1, ..., A_m} *\
begin  old := new;
  while queue <> nil and cons do  new := 0;
    for i := 1 to old and cons do
      c1 := pop(queue);  next := poslit(c1);
      for c2 ∈ H[next].clauselist do  num[c2] := num[c2] − 1;
        if num[c2] = 0 then  k := poslit[c2];
          if H[k].val ≠ 1 then
            if k <> 0 then
              H[k].val := t;  queue := push(c2, queue);  new := new + 1;
            else  cons := f;
        end if end if end if
    end for end for
    old := new;
  end while end
```

Beispiel: Initialisierung

Betrachte folgende Horn-Klauseln:

$$c1: \quad A \qquad\qquad c2: \quad B \vee \neg A \equiv (B \leftarrow A)$$

$$c3: \quad C \vee \neg A \vee \neg B \equiv (C \leftarrow A \wedge B) \qquad c4: \quad \neg B \vee \neg C \equiv (\leftarrow B \wedge C)$$

$$\begin{array}{lcllll} clauselist(A) & = & \{c2, c3\} & poslit(c1) = A & poslit(c3) = C \\ clauselist(B) & = & \{c3, c4\} & poslit(c2) = B & poslit(c4) = 0 \\ clauselist(C) & = & \{c4\} & & \end{array}$$

Es gilt:

$$\begin{array}{ll} \text{partielles Modell } = \{A\} & num(c1) = 0 \\ queue = \{A\} & num(c2) = 1 \\ cons = t & num(c3) = 2 \\ & num(c4) = 2 \end{array}$$

Beispiel: nach 1. While-Loop

$c1: \quad A$

$c2: \quad B \vee \neg A \equiv (B \leftarrow A)$

$c3: \quad C \vee \neg A \vee \neg B \equiv (C \leftarrow A \wedge B)$

$c4: \quad \neg B \vee \neg C \equiv (\leftarrow B \wedge C)$

$$\begin{aligned} clauselist(A) &= \{c2, c3\} \\ clauselist(B) &= \{c3, c4\} \\ clauselist(C) &= \{c4\} \end{aligned}$$

$poslit(c1) = A \qquad poslit(c3) = C$

$poslit(c2) = B \qquad poslit(c4) = 0$

Es gilt:

partielles Modell $= \{A, B\}$

$queue = \{B\}$

$cons = t$

$num(c1) = 0$

$num(c2) = 0$

$num(c3) = 1$

$num(c4) = 2$

Beispiel: nach 2. While-Loop

$c1: \quad A$ $\qquad c2: \quad B \vee \neg A \equiv (B \leftarrow A)$

$c3: \quad C \vee \neg A \vee \neg B \equiv (C \leftarrow A \wedge B)$ $\qquad c4: \quad \neg B \vee \neg C \equiv (\leftarrow B \wedge C)$

$$\begin{array}{lcl} clauselist(A) & = & \{c2, c3\} \\ clauselist(B) & = & \{c3, c4\} \\ clauselist(C) & = & \{c4\} \end{array} \qquad \begin{array}{ll} poslit(c1) = A & poslit(c3) = C \\ poslit(c2) = B & poslit(c4) = 0 \end{array}$$

Es gilt:

partielles Modell $= \{A, B, C\}$ $\qquad num(c1) = 0$

$queue = \{C\}$ $\qquad num(c2) = 0$

$cons = t$ $\qquad num(c3) = 0$

$num(c4) = 1$

Beispiel: nach 3. While-Loop

$$c1: \quad A \qquad\qquad c2: \quad B \vee \neg A \equiv (B \leftarrow A)$$

$$c3: \quad C \vee \neg A \vee \neg B \equiv (C \leftarrow A \wedge B) \qquad c4: \quad \neg B \vee \neg C \equiv (\leftarrow B \wedge C)$$

$$\begin{array}{lcl}
clauselist(A) & = & \{c2, c3\} \\
clauselist(B) & = & \{c3, c4\} \\
clauselist(C) & = & \{c4\}
\end{array}
\qquad
\begin{array}{l}
poslit(c1) = A \\
poslit(c2) = B
\end{array}
\qquad
\begin{array}{l}
poslit(c3) = C \\
poslit(c4) = 0
\end{array}$$

Es gilt:

partielles Modell $= \{A, B, C\}$ $\qquad num(c1) = 0$

$queue = \{0\}$ (false!!) $\qquad num(c2) = 0$

$cons = f$ (!!) $\qquad num(c3) = 0$

$num(c4) = 0$

☞ Inkonsistenz entdeckt

Komplexität von Algorithmus 2

- Jede Horn Klausel $C \in \mathcal{H}$ kommt höchsten einmal in die $queue$; C kommt genau dann in die $queue$ wenn alle ihre negativen Literale t (true) sind.
- Sobald eine Klausel $c2$ bearbeitet wird, wird ihr positives Literal zu t (true), dass verhindert doppelte Behandlung von Klauseln.
- Wenn eine Klausel $c1$ im While-Loop gelöscht wird, werden sofort alle Klauseln behandelt, die das positive Literal von $c1$ negativ enthalten. Jedes negative Vorkommen einer Variable wird nur genau einmal betrachtet. Es gibt nur linear (in n der Anzahlder Klauseln) viele negative Vorkommen von aussagenlogischen Variablen.

☞ Algorithmus 2 braucht $O(n)$ Schritte.

☞ Für definite Programme berechnet Algorithmus 2 das kleinste Herbrand Modell $M_{\mathcal{P}}$

How to Apply the Idea to ASP?

Horn Clauses	normal rules
$a \leftarrow b_1, \ldots, b_n$	$a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_k$
$\leftarrow b_1, \ldots, b_n$	$\leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_k$

Remeber application condition for rule r wrt a set of atoms X

- r has to be supported, that is, $body^+(r) \subseteq X$, and
- r has to be unblocked, that is, $body^-(r) \cap X = \emptyset$.

☞ Use tow counter for each rule, one for support and one for blockage.

Notions of Equivalence

Two programs Π_1 and Π_2 are

- *(weakly) equivalent* ($\Pi_1 \equiv \Pi_2$) if $AS(\Pi_1) = AS(\Pi_2)$.
- *strongly equivalent* ($\Pi_1 \equiv_s \Pi_2$)if $AS(\Pi_1 \cup \Pi') = AS(\Pi_2 \cup \Pi')$ for any program Π' .
- *uniformly equivalent* ($\Pi_1 \equiv_u \Pi_2$) if $AS(\Pi_1 \cup F) = AS(\Pi_2 \cup F)$ for any set of facts F .

Example: $\Pi_1 = \{a \vee b \leftarrow \}$ and $\Pi_2 = \{a \leftarrow \mathit{not}\ b\ , b \leftarrow \mathit{not}\ a\ \}$

- $\Pi_1 \equiv \Pi_2$ since $AS(\Pi_1) = \{\{a\}, \{b\}\} = AS(\Pi_2)$
- $\Pi_1 \equiv_u \Pi_2$
- $\Pi_1 \not\equiv_s \Pi_2$, e.g. $\Pi' = \{a \leftarrow b\ , b \leftarrow a\ \}$

How to Show Strong Equivalence

- close relation to non-classical logic of here-and-there (Lifschitz, Pearce, Valverde)
- model-theoretic characterization (Turner)
  - Let Π be a logic program and X, Y interpretations such that $X \subseteq Y$
  - (X, Y) is an *SE-model* of Π if $Y \models \Pi$ and $X \models \Pi^Y$
  - $M_s(\Pi)$ denotes the set of all SE-models of Π
  - $\Pi_1 \equiv_s \Pi_2$ iff $M_s(\Pi_1) = M_s(\Pi_2)$

How to Show Strong Equivalence (ctd.)

Example: $\Pi_1 = \{a \vee b \leftarrow\ \}$ and $\Pi_2 = \{a \leftarrow not\ b\ , b \leftarrow not\ a\ \}$

$$\begin{aligned} M_s(\Pi_1) = \ & \{(\{a\},\{a\}),\ (\{b\},\{b\}),\ (\{a\},\{a,b\}),\ (\{b\},\{a,b\}), \\ & (\{a,b\},\{a,b\})\} \end{aligned}$$

$$M_s(\Pi_2) = \ M_s(\Pi_1) \cup \{(\emptyset, \{a,b\})\}$$

$M_s(\Pi_1) \neq M_s(\Pi_2)$ therefore $\Pi_1 \not\equiv_s \Pi_2$

How to Show Uniform Equivalence

- model-theoretic characterization (Eiter, Fink)
  - Let Π be a logic program and X, Y interpretations such that $X \subseteq Y$
  - $(X, Y) \in M_s(\Pi)$ is an *UE-model* of Π if
    for every $(X', Y) \in M_s(\Pi)$ it holds that $X \subset X'$ implies $X' = Y$
  - $M_u(\Pi)$ denotes the set of all UE-models of Π
  - $\Pi_1 \equiv_u \Pi_2$ iff $M_u(\Pi_1) = M_u(\Pi_2)$

How to Show Uniform Equivalence (ctd.)

Example: $\Pi_1 = \{a \vee b \leftarrow \}$ and $\Pi_2 = \{a \leftarrow not\ b\ , b \leftarrow not\ a\ \}$

$$\begin{aligned} M_s(\Pi_1) = & \{(\{a\},\{a\}),\ (\{b\},\{b\}),\ (\{a\},\{a,b\}),\ (\{b\},\{a,b\}), \\ & (\{a,b\},\{a,b\})\} \end{aligned}$$

$$M_s(\Pi_2) = M_s(\Pi_1) \cup \{(\emptyset,\{a,b\})\}$$

☞ $(X,Y) = (\emptyset,\{a,b\})$ is not an UE-models of Π_2 since $X \subset X'$ implies $X' = Y$ does not hold e.g. for $(X',Y) = (\{a\},\{a\})$

☞ All other SE-models of Π_1 and Π_2 are also UE-models: $M_u(\Pi_1) = M_u(\Pi_2)$ therefore $\Pi_1 \equiv_u \Pi_2$