Complex Optimization in Answer Set Programming

Martin Gebser Roland Kaminski Torsten Schaub

University of Potsdam

Outline

Outline

Motivation

Fact

Preferences and Optimization play an important role in Knowledge Representation and many other domains!

- Indeed, systems like *dlv*, *clasp*, or *smodels* provide
  - optimization statements
  - weak constraints

  for expressing cost functions on sets of weighted literals
- Also, pre-processors exist for expressing preferences among rules
- No readily applicable implementation techniques for qualitative preferences among answer sets, like
  - inclusion minimality,
  - Pareto efficiency, or
  - complex preference languages

Motivation

Fact

Preferences and Optimization play an important role in Knowledge Representation and many other domains!

- Indeed, systems like *dlv*, *clasp*, or *smodels* provide
  - optimization statements
  - weak constraints

  for expressing cost functions on sets of weighted literals
- Also, pre-processors exist for expressing preferences among rules
- No readily applicable implementation techniques for qualitative preferences among answer sets, like
  - inclusion minimality,
  - Pareto efficiency, or
  - complex preference languages

Motivation

Fact

Preferences and Optimization play an important role in Knowledge Representation and many other domains!

- Indeed, systems like *dlv*, *clasp*, or *smodels* provide
  - optimization statements
  - weak constraints

  for expressing cost functions on sets of weighted literals
- Also, pre-processors exist for expressing preferences among rules
- No readily applicable implementation techniques for qualitative preferences among answer sets, like
  - inclusion minimality,
  - Pareto efficiency, or
  - complex preference languages

Are complex preferences needed?

- Complex preferences are vital in
  - Argumentation
  - Belief Change
  - Bio-informatics
  - Circumscription
  - Decision Making
  - Diagnosis
  - Inconsistency Handling
  - System Design
  - etc.

- *How come there are no readily available implementation techniques for complex preferences?*

Are complex preferences needed?

- Complex preferences are vital in
  - Argumentation
  - Belief Change
  - Bio-informatics
  - Circumscription
  - Decision Making
  - Diagnosis
  - Inconsistency Handling
  - System Design
  - etc.

- *How come there are no readily available implementation techniques for complex preferences?*

Are complex preferences needed?

- Complex preferences are vital in
  - Argumentation
  - Belief Change
  - Bio-informatics
  - Circumscription
  - Decision Making
  - Diagnosis
  - Inconsistency Handling
  - System Design
  - etc.

- *How come there are no readily available implementation techniques for complex preferences?*

What makes complex preferences intricate?

- Complex preferences lead to a significant increase in computational complexity for normal logic programs
- Because they combine an NP with a coNP problem, where
  - the first one defines feasible solutions
  - the second one ensures that there are no better solutions

- For implementing such problems, Eiter and Gottlob invented the *saturation technique*, using the elevated complexity of disjunctive logic programming

- BUT: this technique is rather involved and hardly usable by ASP laymen

What makes complex preferences intricate?

- Complex preferences lead to a significant increase in computational complexity for normal logic programs
- Because they combine an NP with a coNP problem, where
  - the first one defines feasible solutions
  - the second one ensures that there are no better solutions

- For implementing such problems, Eiter and Gottlob invented the *saturation technique*, using the elevated complexity of disjunctive logic programming

- BUT: this technique is rather involved and hardly usable by ASP laymen

What makes complex preferences intricate?

- Complex preferences lead to a significant increase in computational complexity for normal logic programs
- Because they combine an NP with a coNP problem, where
  - the first one defines feasible solutions
  - the second one ensures that there are no better solutions

- For implementing such problems, Eiter and Gottlob invented the *saturation technique*, using the elevated complexity of disjunctive logic programming

- BUT: this technique is rather involved and hardly usable by ASP laymen

What makes complex preferences intricate?

- Complex preferences lead to a significant increase in computational complexity for normal logic programs
- Because they combine an NP with a coNP problem, where
  - the first one defines feasible solutions
  - the second one ensures that there are no better solutions

- For implementing such problems, Eiter and Gottlob invented the *saturation technique*, using the elevated complexity of disjunctive logic programming

- BUT: this technique is rather involved and hardly usable by ASP laymen

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}, \{a, c\}, \{a, b, c\}$ vs $\{a^f, b^t, c^f\}, \{a^t, b^f, c^t\}, \{a^t, b^t, c^t\}$

Saturation

1. Define viable solution candidates (extended logic program)
2. Define viable counterexamples (disjunctive logic program)
3. Spoil invalid counterexamples and query

$\subseteq$-minimal models of $\{a \vee b, a \vee \neg b \vee \neg c, \neg a \vee c\}$

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

```
a^t | a^f.  b^t | b^f.  c^t | c^f.
⊥ :- a^f, b^f.
⊥ :- a^f, b^t, c^t.
⊥ :- a^t, c^f.
a^t :- ⊥.  a^f :- ⊥.
b^t :- ⊥.  b^f :- ⊥.
c^t :- ⊥.  c^f :- ⊥.
⊥ :- a^t, not a.  ā :- a^t.  ā :- not a.
⊥ :- b^t, not b.  b̄ :- b^t.  b̄ :- not b.
⊥ :- c^t, not c.  c̄ :- c^t.  c̄ :- not c.
⊥ :- ā, b̄, c̄.
:- not ⊥.
```

Answer sets

$\{b\}$, $\{a, c\}$, $\{a, b, c\}$ with $\{a^t, a^f, b^t, b^f, c^t, c^f, \bar{a}, \bar{b}, \bar{c}, \bot\}$

Outline

Preliminaries

- A program consists of rules of the form

  $$\texttt{H :- B}_1, \ldots, \texttt{B}_m, \texttt{not B}_{m+1}, \ldots, \texttt{not B}_n.$$

  where each $\texttt{H}$ and $\texttt{B}_i$ is either
  - an *atom* $\texttt{a}_i$ or
  - a $\#\texttt{count}$ *constraint*

    $$\texttt{L } \{\ell_1, \ldots, \ell_k\} \texttt{ U}$$

    where $\ell_i = \texttt{a}_i$ or $\ell_i = \texttt{not a}_i$ is a *literal*
- and possibly some $\#\texttt{minimize}$ statements of the form

  $$\#\texttt{minimize } \{\ell_1, \ldots, \ell_k\}.$$

- Note: our approach supports the full input language of *gringo*.

Preliminaries

- A program consists of rules of the form

  $$\texttt{H :- B}_1, \ldots, \texttt{B}_m, \texttt{not B}_{m+1}, \ldots, \texttt{not B}_n.$$

  where each $\texttt{H}$ and $\texttt{B}_i$ is either
  - an *atom* $\texttt{a}_i$ or
  - a $\#\texttt{count}$ *constraint*

    $$\texttt{L}\ \{\ell_1, \ldots, \ell_k\}\ \texttt{U}$$

    where $\ell_i = \texttt{a}_i$ or $\ell_i = \texttt{not a}_i$ is a *literal*
- and possibly some $\#\texttt{minimize}$ statements of the form

  $$\#\texttt{minimize}\ \{\ell_1, \ldots, \ell_k\}.$$

- Note: our approach supports the full input language of *gringo*.

Preliminaries

- A program consists of rules of the form

  $$\texttt{H :- B}_1, \ldots, \texttt{B}_m, \texttt{not B}_{m+1}, \ldots, \texttt{not B}_n.$$

  where each $\texttt{H}$ and $\texttt{B}_i$ is either
  - an *atom* $\texttt{a}_i$ or
  - a $\#\texttt{count}$ *constraint*

    $$\texttt{L } \{\ell_1, \ldots, \ell_k\} \texttt{ U}$$

    where $\ell_i = \texttt{a}_i$ or $\ell_i = \texttt{not a}_i$ is a *literal*
- and possibly some $\#\texttt{minimize}$ statements of the form

  $$\#\texttt{minimize } \{\ell_1, \ldots, \ell_k\}.$$

- Note: our approach supports the full input language of *gringo*.

An example without optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
       s     :- not q, not r.
```

```
gringo example.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p s
Answer: 3
p s t
Answer: 4
p q
Answer: 5
p r

Models       : 5
Time         : 0.000s
CPU Time     : 0.000s
```

We get 5 answer sets

- $\{s, t\}$
- $\{p, s\}$
- $\{p, s, t\}$
- $\{p, q\}$
- $\{p, r\}$

An example without optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
```

```
gringo example.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p s
Answer: 3
p s t
Answer: 4
p q
Answer: 5
p r

Models       : 5
Time         : 0.000s
CPU Time     : 0.000s
```

We get 5 answer sets

- $\{s, t\}$
- $\{p, s\}$
- $\{p, s, t\}$
- $\{p, q\}$
- $\{p, r\}$

An example with optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

```
gringo example.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
s t
Optimization: 1
OPTIMUM FOUND

Models       : 1
  Optimum    : yes
Time         : 0.000s
CPU Time     : 0.000s
```

We get 1 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- ~~$\{p, q\}$~~
- ~~$\{p, r\}$~~

An example with optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

gringo example.lp | clasp 0

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
s t
Optimization: 1
OPTIMUM FOUND

Models       : 1
  Optimum    : yes
Time         : 0.000s
CPU Time     : 0.000s
```

We get 1 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- ~~$\{p, q\}$~~
- ~~$\{p, r\}$~~

Wanted: $\subseteq$-based optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

```
mytool example.lp | gringo | clasp  0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p q
Answer: 3
p r
SATISFIABLE

Models       : 3
Time         : 0.000s
CPU Time     : 0.000s
```

We want 3 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- $\{p, q\}$
- $\{p, r\}$

Wanted: $\subseteq$-based optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

```
mytool example.lp | gringo | claspD 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p q
Answer: 3
p r
SATISFIABLE

Models       : 3
Time         : 0.000s
CPU Time     : 0.000s
```

We want 3 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- $\{p, q\}$
- $\{p, r\}$

Wanted: $\subseteq$-based optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

```
mytool example.lp | gringo | claspD 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p q
Answer: 3
p r
SATISFIABLE

Models       : 3
Time         : 0.000s
CPU Time     : 0.000s
```

We want 3 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- $\{p, q\}$
- $\{p, r\}$

Wanted: $\subseteq$-based optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

```
mytool example.lp | gringo | claspD 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p q
Answer: 3
p r
SATISFIABLE

Models       : 3
Time         : 0.000s
CPU Time     : 0.000s
```

We want 3 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- $\{p, q\}$
- $\{p, r\}$

Wanted: $\subseteq$-based optimization

example.lp

```
1 { p, t }   :- 1 { r, s, not t } 2.
  { q, r } 1 :- 1 { p, t }.
    s        :- not q, not r.
#minimize{ p, q, r, s }.
```

```
mytool example.lp | gringo | claspD 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
s t
Answer: 2
p q
Answer: 3
p r
SATISFIABLE

Models       : 3
Time         : 0.000s
CPU Time     : 0.000s
```

We want 3 answer sets

- $\{s, t\}$
- ~~$\{p, s\}$~~
- ~~$\{p, s, t\}$~~
- $\{p, q\}$
- $\{p, r\}$

Outline

Grounding by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

Grounding by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo easy.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
p(1) p(2) p(3) q(2)
SATISFIABLE

Models       : 1
Time         : 0.000s
CPU Time     : 0.000s
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo easy.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
p(1) p(2) p(3) q(2)
SATISFIABLE

Models       : 1
Time         : 0.000s
CPU Time     : 0.000s
```

Reifying by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)   : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                   not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)   : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                   not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
           not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
           not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
           not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
           not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
           not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                          not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                          L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                     not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
           not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
#show hold(atom(p(1))). #show hold(atom(p(2))). #show hold(atom(p(3))). #show hold(atom(q(2))).
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
#show hold(atom(p(1))). #show hold(atom(p(2))). #show hold(atom(p(3))). #show hold(atom(q(2))).
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
#show hold(atom(p(1))). #show hold(atom(p(2))). #show hold(atom(p(3))). #show hold(atom(q(2))).
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
#show hold(atom(p(1))). #show hold(atom(p(2))). #show hold(atom(p(3))). #show hold(atom(q(2))).
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --reify easy.lp | gringo - meta.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
hold(atom(p(3))) hold(atom(p(2))) hold(atom(p(1))) hold(atom(q(2)))
SATISFIABLE

Models       : 1
Time         : 0.000s
CPU Time     : 0.000s
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --reify easy.lp | gringo - meta.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
hold(atom(p(3))) hold(atom(p(2))) hold(atom(p(1))) hold(atom(q(2)))
SATISFIABLE

Models       : 1
Time         : 0.000s
CPU Time     : 0.000s
```

Outline

Idea

- An answer set generated via `meta.lp` is optimal, only if it is not dominated by any other answer set
- Use disjunctive ASP to represent all potential counterexamples
- Encode the subtasks of
  1. guessing an answer set as a potential counterexample and
  2. verifying that the counterexample dominates a candidate answer set
- A candidate answer set passes both phases, if it is inpossible to guess a counterexample that dominates it

- In all, we get three meta-encodings
  - `meta.lp` as above
  - `metaD.lp` for guessing a potential counterexample
  - `metaO.lp` for verifying non-dominance

Idea

- An answer set generated via `meta.lp` is optimal, only if it is not dominated by any other answer set
- Use disjunctive ASP to represent all potential counterexamples
- Encode the subtasks of
  1. guessing an answer set as a potential counterexample and
  2. verifying that the counterexample dominates a candidate answer set
- A candidate answer set passes both phases, if it is inpossible to guess a counterexample that dominates it

- In all, we get three meta-encodings
  - `meta.lp` as above
  - `metaD.lp` for guessing a potential counterexample
  - `metaO.lp` for verifying non-dominance

metaD.lp

- Encodes answer sets of extended logic programs by a positive disjunctive meta-program, including rules for
  1. guessing an interpretation $\mathcal{I}$:
     `true(atom(A)) | fail(atom(A)) :- elem(atom(A)).`
  2. deriving error-indicating atom `bot` if $\mathcal{I}$ is not supported model
  3. deriving `bot` if true atoms (of SCCs) are not acyclicly derivable
  4. deriving all truth assignments for atoms from `bot` (saturation):
     `true(atom(A)) :- elem(atom(A)), bot.`
     `fail(atom(A)) :- elem(atom(A)), bot.`
- Encodes acyclic derivability checks via
  - complement of immediate consequence operator
  - without guessing any level mapping (cf. [Eiter & Polleres, 06])
- One (proper) disjunctive meta-rule!
  - given in

metaD.lp

- Encodes answer sets of extended logic programs by a positive disjunctive meta-program, including rules for
  1. guessing an interpretation $\mathcal{I}$:

     ```
     true(atom(A)) | fail(atom(A)) :- elem(atom(A)).
     ```

  2. deriving error-indicating atom `bot` if $\mathcal{I}$ is not supported model
  3. deriving `bot` if true atoms (of SCCs) are not acyclicly derivable
  4. deriving all truth assignments for atoms from `bot` (saturation):

     ```
     true(atom(A)) :- elem(atom(A)), bot.
     fail(atom(A)) :- elem(atom(A)), bot.
     ```

- Encodes acyclic derivability checks via
  - complement of immediate consequence operator
  - without guessing any level mapping (cf. [Eiter & Polleres, 06])
- One (proper) disjunctive meta-rule!
  - given in 1

metaD.lp

- Encodes answer sets of extended logic programs by a positive disjunctive meta-program, including rules for
  1. guessing an interpretation $\mathcal{I}$:

     ```
     true(atom(A)) | fail(atom(A)) :- elem(atom(A)).
     ```

  2. deriving error-indicating atom `bot` if $\mathcal{I}$ is not supported model
  3. deriving `bot` if true atoms (of SCCs) are not acyclicly derivable
  4. deriving all truth assignments for atoms from `bot` (saturation):

     ```
     true(atom(A)) :- elem(atom(A)), bot.
     fail(atom(A)) :- elem(atom(A)), bot.
     ```

- Encodes acyclic derivability checks via
  - complement of immediate consequence operator
  - without guessing any level mapping (cf. [Eiter & Polleres, 06])
- One (proper) disjunctive meta-rule!
  - given in 1

metaD.lp

- Encodes answer sets of extended logic programs by a positive disjunctive meta-program, including rules for
  1. guessing an interpretation $\mathcal{I}$:

     ```
     true(atom(A)) | fail(atom(A)) :- elem(atom(A)).
     ```

  2. deriving error-indicating atom `bot` if $\mathcal{I}$ is not supported model
  3. deriving `bot` if true atoms (of SCCs) are not acyclicly derivable
  4. deriving all truth assignments for atoms from `bot` (saturation):

     ```
     true(atom(A)) :- elem(atom(A)), bot.
     fail(atom(A)) :- elem(atom(A)), bot.
     ```

- Encodes acyclic derivability checks via
  - complement of immediate consequence operator
  - without guessing any level mapping (cf. [Eiter & Polleres, 06])
- One (proper) disjunctive meta-rule!
  - given in 1

metaO.lp

- Encodes user-defined preferences composed from
  - cardinality minimality
  - inclusion minimality
  - literal preferences (cf. [Sakama & Inoue, 00])
- Aggregation via lexicographic ranking and/or Pareto efficiency
- Given a statement of the form

  `#minimize[` $\ell_1 = w_1 @ L_1$ `,` $\ldots$ `,` $\ell_k = w_k @ L_k$ `].`
  - priority levels L_i are ordered lexicographically: $1 < 2 < 3 < \ldots$
  - weights w_i distinguish literals at a level: $1@L \parallel 2@L \parallel 3@L \parallel \ldots$
  - optimization criteria by level and weight customizable via facts:
    `optimize(`L`,` w`, card;incl;pref).`
- Comparison(s) of candidate answer set and counterexample
  - encoded by stratified logic program
  - deriving error-indicating atom `bot` if candidate not dominated (triggering saturation implemented in `metaD.lp`)
  - candidates admitting counterexample refuted by `:- not bot.`

`metaO.lp`

- Encodes user-defined preferences composed from
  - cardinality minimality
  - inclusion minimality
  - literal preferences (cf. [Sakama & Inoue, 00])
- Aggregation via lexicographic ranking and/or Pareto efficiency
- Given a statement of the form

  `#minimize[` $\ell_1 = w_1$`@`L_1`,` $\ldots$`,` $\ell_k = w_k$`@`L_k `].`
  - priority levels L_i are ordered lexicographically: $1 < 2 < 3 < \ldots$
  - weights w_i distinguish literals at a level: $1@L \parallel 2@L \parallel 3@L \parallel \ldots$
  - optimization criteria by level and weight customizable via facts:
    `optimize(`L`,` w`, card;incl;pref).`
- Comparison(s) of candidate answer set and counterexample
  - encoded by stratified logic program
  - deriving error-indicating atom `bot` if candidate not dominated (triggering saturation implemented in `metaD.lp`)
  - candidates admitting counterexample refuted by `:- not bot.`

metaO.lp

- Encodes user-defined preferences composed from
  - cardinality minimality
  - inclusion minimality
  - literal preferences (cf. [Sakama & Inoue, 00])
- Aggregation via lexicographic ranking and/or Pareto efficiency
- Given a statement of the form

  `#minimize[` $\ell_1 = w_1$`@`L_1`,` $\ldots$`,` $\ell_k = w_k$`@`L_k `].`
  - priority levels L_i are ordered lexicographically: $1 < 2 < 3 < \ldots$
  - weights w_i distinguish literals at a level: $1@L \parallel 2@L \parallel 3@L \parallel \ldots$
  - optimization criteria by level and weight customizable via facts:

    `optimize(`L`,` w`, card;incl;pref).`
- Comparison(s) of candidate answer set and counterexample
  - encoded by stratified logic program
  - deriving error-indicating atom `bot` if candidate not dominated (triggering saturation implemented in `metaD.lp`)
  - candidates admitting counterexample refuted by `:- not bot.`

metasp

- http://www.cs.uni-potsdam.de/wv/metasp
  also via http://potassco.sourceforge.net/labs.html

- Cardinality-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,card).") | \
       claspD 0
```

- Inclusion-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,incl).") | \
       claspD 0
```

- Pareto-efficient answer sets

```
gringo --reify examples/p1.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} | \
       claspD 0
```

- And combinations thereof!

metasp

- http://www.cs.uni-potsdam.de/wv/metasp
  also via http://potassco.sourceforge.net/labs.html

- Cardinality-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,card).") | \
       claspD 0
```

- Inclusion-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,incl).") | \
       claspD 0
```

- Pareto-efficient answer sets

```
gringo --reify examples/p1.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} | \
       claspD 0
```

- And combinations thereof!

metasp

- http://www.cs.uni-potsdam.de/wv/metasp
  also via http://potassco.sourceforge.net/labs.html

- Cardinality-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,card).") | \
       claspD 0
```

- Inclusion-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,incl).") | \
       claspD 0
```

- Pareto-efficient answer sets

```
gringo --reify examples/p1.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} | \
       claspD 0
```

- And combinations thereof!

metasp

- http://www.cs.uni-potsdam.de/wv/metasp
  also via http://potassco.sourceforge.net/labs.html

- Cardinality-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,card).") | \
       claspD 0
```

- Inclusion-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,incl).") | \
       claspD 0
```

- Pareto-efficient answer sets

```
gringo --reify examples/p1.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} | \
       claspD 0
```

- And combinations thereof!

metasp

- http://www.cs.uni-potsdam.de/wv/metasp
  also via http://potassco.sourceforge.net/labs.html

- Cardinality-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,card).") | \
       claspD 0
```

- Inclusion-minimal answer sets

```
gringo --reify examples/p0.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} <(echo "optimize(1,1,incl).") | \
       claspD 0
```

- Pareto-efficient answer sets

```
gringo --reify examples/p1.lp | \
       gringo - encodings/{meta.lp,metaD.lp,metaO.lp} | \
       claspD 0
```

- And combinations thereof!

Outline

Some experiments

repair mode	direct encoding time	direct encoding timeout	meta encoding time	meta encoding timeout
mode 1	315801	72	4823	0
mode 2	45736	0	42308	2
mode 3	365227	78	366798	79
$\sum$	726764	150	413929	81

- Benchmarks: 3×100 random samples from real data of two biological experiments, used in [KR'10]
- Systems: *gringo* 3.0.3 and *claspD* 1.1
- Settings: timout of 4000 seconds
- Observation: Meta-programming solved 219 versus 150 !

Some experiments

repair mode	direct encoding time	direct encoding timeout	meta encoding time	meta encoding timeout
mode 1	315801	72	4823	0
mode 2	45736	0	42308	2
mode 3	365227	78	366798	79
$\sum$	726764	150	413929	81

- Benchmarks: 3×100 random samples from real data of two biological experiments, used in [KR'10]
- Systems: *gringo* 3.0.3 and *claspD* 1.1
- Settings: timout of 4000 seconds
- Observation: Meta-programming solved 219 versus 150 !

Some experiments

repair mode	direct encoding time	direct encoding timeout	meta encoding time	meta encoding timeout
mode 1	315801	72	4823	0
mode 2	45736	0	42308	2
mode 3	365227	78	366798	79
$\sum$	726764	150	413929	81

- Benchmarks: 3×100 random samples from real data of two biological experiments, used in [KR'10]
- Systems: *gringo* 3.0.3 and *claspD* 1.1
- Settings: timout of 4000 seconds
- Observation: Meta-programming solved 219 versus 150 !

Outline

Summary

- Extended the ease of ASP modeling to complex preferences (involving a higher computational complexity)
- A meta-programming approach to complex optimization is an eligible and viable alternative

- Need enhancements of disjunctive ASP solvers

- *This is a stronghold of ASP!*

Summary

- Extended the ease of ASP modeling to complex preferences (involving a higher computational complexity)
- A meta-programming approach to complex optimization is an eligible and viable alternative

- Need enhancements of disjunctive ASP solvers

- *This is a stronghold of ASP!*

Summary

- Extended the ease of ASP modeling to complex preferences (involving a higher computational complexity)
- A meta-programming approach to complex optimization is an eligible and viable alternative

- Need enhancements of disjunctive ASP solvers

- *This is a stronghold of ASP!*