Answer Set Solving in Practice

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

Multi-shot ASP Solving: Overview

Outline

Potassco

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*

  Multi-shot solving: *ground* | *solve*

  ➥ **continuously changing logic programs**

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

Potassco

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*

  Multi-shot solving: *ground* | *solve*

  ➥ **continuously changing logic programs**

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: *ground* | *solve*

  ➥ **continuously changing logic programs**

- Application areas

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: *ground* | *solve*

  ➥ **continuously changing logic programs**

- Application areas

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

Potassco

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: *ground** | *solve**

  ➥ **continuously changing logic programs**

- Application areas

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*ground** | *solve**)*

  ➥ **continuously changing logic programs**

- Application areas

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**)*
  - ➥ **continuously changing logic programs**

- Application areas

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory*)*
  - ➥ **continuously changing logic programs**
- Application areas

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | ...)*
  - ➥ **continuously changing logic programs**

- Application areas

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

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*

  ➥ **continuously changing logic programs**

- Application areas

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

Potassco

Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | ...)*

  ➥ **continuously changing logic programs**

- Application areas

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

Clingo = ASP + Control

- ASP

  `#program <name> [ (<parameters>) ]`

  `#program play(t).`

  `#external <atom> [ : <body> ]`

  `#external mark(X,Y,P,t) : field(X,Y), player(P).`

- Control

  Python (`www.python.org`)

  `prg.solve(), prg.ground(parts), ...`

  C, Lua, and Prolog embeddings are available too

- Integration

  in ASP: embedded scripting language (`#script`)

  in Python: library import (`import clingo`)

Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Python (`www.python.org`)
    - `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Python (`www.python.org`)

    `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

Potassco

Clingo $=$ ASP $+$ Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

Clingo = ASP + Control

- ASP
  - #program <name> [(<parameters>)]
    - Example #program play(t).
  - #external <atom> [: <body>]
    - Example #external mark(X,Y,P,t) : field(X,Y), player(P).

- Control
  - Python (www.python.org)
    - Example prg.solve(), prg.ground(parts), ...
  - C, Lua, and Prolog embeddings are available too

- Integration
  - in ASP: embedded scripting language (#script)
  - in Python: library import (import clingo)

Clingo $=$ ASP $+$ Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

Vanilla *clingo*

```
#script (python)
def main(prg):
    parts = []
    parts.append(("base", []))
    prg.ground(parts)
    prg.solve()
#end.
```

Vanilla *clingo*

```
#script (python)
def main(prg):
    parts = []
    parts.append(("base", []))
    prg.ground(parts)
    prg.solve()
#end.
```

Potassco

Vanilla *clingo*

```
#script (python)
def main(prg):
    parts = []
    parts.append(("base", []))
    prg.ground(parts)
    prg.solve()
#end.
```

Hello world!

```
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

```
$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models       : 0+
Calls        : 1
Time         : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Hello world!

```
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

```
$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models       : 0+
Calls        : 1
Time         : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Hello world!

```
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

```
$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models       : 0+
Calls        : 1
Time         : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Hello world!

```
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

```
$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models       : 0+
Calls        : 1
Time         : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Outline

#program declaration

- A program declaration is of form

  $\texttt{\#program}\ n\,(p_1, \dots, p_k)$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

#program declaration

- A program declaration is of form

  #program $n\,(p_1, \ldots, p_k)$

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Potassco

#program declaration

- A program declaration is of form

  `#program` $n\,(p_1,\dots,p_k)$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

#program declaration

- A program declaration is of form

  $$\texttt{\#program}\ n(p_1,\dots,p_k)$$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

#program declaration

- A program declaration is of form

  $$\texttt{\#program}\ n(p_1, \dots, p_k)$$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Potassco

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration

- Example

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

$$R(n)[p_1/t_1, \dots, p_k/t_k]$$

obtained by replacing in $R(n)$ each occurrence of p_i by t_i

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

  $$R(n)[p_1/t_1, \dots, p_k/t_k]$$

  obtained by replacing in $R(n)$ each occurrence of p_i by t_i

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

  $$R(n)[p_1/t_1, \ldots, p_k/t_k]$$

  obtained by replacing in $R(n)$ each occurrence of p_i by t_i

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

  $$R(n)[p_1/t_1, \ldots, p_k/t_k]$$

  obtained by replacing in $R(n)$ each occurrence of p_i by t_i

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid})[k/42] = \{b(k), c(X,k) \leftarrow a(X)\}[k/42]$
- Given a name n with associated parameters $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

  $$R(n)[p_1/t_1, \dots, p_k/t_k]$$

  obtained by replacing in $R(n)$ each occurrence of p_i by t_i

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid})[k/42] = \{b(42), c(X, 42) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

  $$R(n)[p_1/t_1, \ldots, p_k/t_k]$$

  obtained by replacing in $R(n)$ each occurrence of p_i by t_i

Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base
- Given a set R of (non-ground) rules and two sets C, D of ground atoms, we define an instantiation of R relative to C as a ground program $ground_C(R)$ over D subject to the following conditions:

$$C \subseteq D \subseteq C \cup head(ground_C(R))$$

$$ground_C(R) \subseteq \{head(r) \leftarrow body(r)^+ \cup \{\sim a \mid a \in body(r)^- \cap D\} \mid r \in ground(R), body(r)^+ \subseteq D\}$$

- Example Given $R = \{\ a(X) \leftarrow f(X), e(X);\ \ b(X) \leftarrow f(X), \sim e(X)\ \}$ and $C = \{f(1), f(2), e(1)\}$, we obtain

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

Potassco

Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base
- Given a set R of (non-ground) rules and two sets C, D of ground atoms, we define an instantiation of R relative to C as a ground program $ground_C(R)$ over D subject to the following conditions:

$$C \subseteq D \subseteq C \cup head(ground_C(R))$$

$$ground_C(R) \subseteq \{head(r) \leftarrow body(r)^+ \cup \{\sim a \mid a \in body(r)^- \cap D\} \mid r \in ground(R), body(r)^+ \subseteq D\}$$

- Example Given $R = \{\ a(X) \leftarrow f(X), e(X);\ \ b(X) \leftarrow f(X), \sim e(X)\ \}$ and $C = \{f(1), f(2), e(1)\}$, we obtain

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

Potassco

Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base
- Given a set R of (non-ground) rules and two sets C, D of ground atoms, we define an instantiation of R relative to C as a ground program $ground_C(R)$ over D subject to the following conditions:

$$C \subseteq D \subseteq C \cup head(ground_C(R))$$

$$ground_C(R) \subseteq \{head(r) \leftarrow body(r)^+ \cup \{\sim a \mid a \in body(r)^- \cap D\} \mid r \in ground(R), body(r)^+ \subseteq D\}$$

- Example Given $R = \{\ a(X) \leftarrow f(X), e(X);\ \ b(X) \leftarrow f(X), \sim e(X)\ \}$ and $C = \{f(1), f(2), e(1)\}$, we obtain

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

#external declaration

- An external declaration is of form

  `#external` $a : B$

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

#external declaration

- An external declaration is of form

  `#external` $a : B$

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Potassco

#external declaration

- An external declaration is of form

  #external $a : B$

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Grounding extensible logic programs

- Given an extensible program R, we define

$$Q = \{a \leftarrow B, \varepsilon \mid (\texttt{\#external}\ a : B) \in R\}$$
$$R' = \{a \leftarrow B \in R\}$$

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

$$P = \{r \in ground_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin body(r)\}$$
$$E = \{head(r) \mid r \in ground_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in body(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup head(P)$

Grounding extensible logic programs

- Given an extensible program R, we define

$$Q = \{a \leftarrow B, \varepsilon \mid (\texttt{\#external}\ a : B) \in R\}$$
$$R' = \{a \leftarrow B \in R\}$$

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

$$P = \{r \in ground_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin body(r)\}$$
$$E = \{head(r) \mid r \in ground_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in body(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup head(P)$

Grounding extensible logic programs

- Given an extensible program R, we define

$$Q = \{a \leftarrow B, \varepsilon \mid (\texttt{\#external}\ a : B) \in R\}$$
$$R' = \{a \leftarrow B \in R\}$$

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

$$P = \{r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin \mathit{body}(r)\}$$
$$E = \{\mathit{head}(r) \mid r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in \mathit{body}(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup \mathit{head}(P)$

Grounding extensible logic programs

- Given an extensible program R, we define

$$Q = \{a \leftarrow B, \varepsilon \mid (\texttt{\#external}\ a : B) \in R\}$$
$$R' = \{a \leftarrow B \in R\}$$

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

$$P = \{r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin \mathit{body}(r)\}$$
$$E = \{\mathit{head}(r) \mid r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in \mathit{body}(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup \mathit{head}(P)$

Example

- Extensible program

```
#external e(X) : f(X), g(X).
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).    b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(X) :- f(X), g(X), ε.
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Atom base $\{g(1)\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).    b(2) :- f(2).
```

with externals $\{e(1)\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.    e(2) :- f(2), g(2), ε.
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).    b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.    e(2) :- f(2), g(2), ε.
f(1). f(2).
a(1) :- f(1), e(1).       a(2) :- f(2), e(2).
b(1) :- f(1), not e(1).   b(2) :- f(2), not e(2).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.   e(2) :- f(2), g(2), ε.
f(1). f(2).
a(1) :- f(1), e(1).      a(2) :- f(2), e(2).
b(1) :- f(1), not e(1).  b(2) :- f(2), not e(2).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.    e(2) :- f(2), g(2), ε.
f(1). f(2).
a(1) :- f(1), e(1).       a(2) :- f(2), e(2).
b(1) :- f(1), not e(1).   b(2) :- f(2), not e(2).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.    e(2) :- f(2), g(2), ε.
f(1). f(2).
a(1) :- f(1), e(1).       a(2) :- f(2), e(2).
b(1) :- f(1), not e(1).   b(2) :- f(2), not e(2).
```

  Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

  with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).  b(2) :- f(2).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).  b(2) :- f(2).
```

Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $\{$e(1)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).  b(2) :- f(2).
```

  Atom base $\{$g(1)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- e(1).
b(1) :- not e(1).          b(2).
```

  with externals $\{$e(1)$\}$

Outline

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:

- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $\mathit{atom}(P) \subseteq I \cup O$, and
    - $\mathit{head}(P) \subseteq O$

- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$ for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$

  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q})\,,\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P}))\,,\ O(\mathbb{P}) \cup O(\mathbb{Q})\ )$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if

  $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and

  $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$

  for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$

  Recursion between two modules to be joined is disallowed

  Recursion within each module is allowed

The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\; P(\mathbb{P}) \cup P(\mathbb{Q}) \, , \; (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \, , \; O(\mathbb{P}) \cup O(\mathbb{Q}) \;)$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$ for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q}) \ , \ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \ , \ O(\mathbb{P}) \cup O(\mathbb{Q}) \ )$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\, P(\mathbb{P}) \cup P(\mathbb{Q}) \, , \, (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \, , \, O(\mathbb{P}) \cup O(\mathbb{Q}) \,)$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\; P(\mathbb{P}) \cup P(\mathbb{Q}) \,, \; (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \,, \; O(\mathbb{P}) \cup O(\mathbb{Q}) \;)$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q}) \, , \ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \, , \ O(\mathbb{P}) \cup O(\mathbb{Q}) \ )$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q}) \, , \ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \, , \ O(\mathbb{P}) \cup O(\mathbb{Q}) \ )$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$$

  via the ground program P with externals E obtained from R and C
- Note $E \setminus head(P)$ consists of atoms stemming from non-overwritten external declarations

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$$

via the ground program P with externals E obtained from R and C

- Note $E \setminus head(P)$ consists of atoms stemming from non-overwritten external declarations

Composing logic programs with externals

- **Idea** Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$$

via the ground program P with externals E obtained from R and C

- **Note** $E \setminus head(P)$ consists of atoms stemming from non-overwritten external declarations

Example

- Atom base $C = \{$`g(1)`$\}$
- Extensible program R

```
#external e(X) : f(X), g(X)
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Ground program P

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).    b(2) :- f(2).
```

with externals $E = \{\texttt{e(1)}\}$

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Potassco

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Ground program P

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $E = \{\texttt{e(1)}\}$

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), \sim e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Extensible program R

```
#external e(X) : f(X), g(X)
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

  $$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

  $$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$
- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

  $$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

  $$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$
- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base

- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$
- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Let $(R_i)_{i>0}$ be a sequence of (non-ground) extensible programs, and let P_{i+1} be the ground program with externals E_{i+1} obtained from R_{i+1} and $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

If $\bigsqcup_{i\geq 0} \mathbb{P}_i$ is compositional, then

1. $P(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} P_i$
2. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} head(P_i)$
3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} head(P_i)$

Capturing program states by modules

- Let $(R_i)_{i>0}$ be a sequence of (non-ground) extensible programs, and let P_{i+1} be the ground program with externals E_{i+1} obtained from R_{i+1} and $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

  If $\bigsqcup_{i\geq 0} \mathbb{P}_i$ is compositional, then

  1. $P(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} P_i$
  2. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} head(P_i)$
  3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} head(P_i)$

Outline

Clingo state

- A *clingo* state is a triple

$$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R}$ is a collection of extensible (non-ground) logic programs
  - $\mathbb{P}$ is a module
  - V is a three-valued assignment over $I(\mathbb{P})$

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - V is a three-valued assignment over $I(\mathbb{P})$

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - $V = (V^t, V^u)$ is a three-valued assignment over $I(\mathbb{P})$ where $V^f = I(\mathbb{P}) \setminus (V^t \cup V^u)$

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - $V = (V^t, V^u)$ is a three-valued assignment over $I(\mathbb{P})$ where $V^f = I(\mathbb{P}) \setminus (V^t \cup V^u)$

- Note Input atoms in $I(\mathbb{P})$ are taken to be false by default

create

- $\mathit{create}(R) : \;\mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

create

- $\mathit{create}(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

Potassco

add

- $add(R) : (\boldsymbol{R}_1, \mathbb{P}, V) \mapsto (\boldsymbol{R}_2, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R}_1 = (R_c)_{c \in \mathcal{C}}$ and $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

Potassco

add

- $add(R) : (\boldsymbol{R}_1, \mathbb{P}, V) \mapsto (\boldsymbol{R}_2, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R}_1 = (R_c)_{c \in \mathcal{C}}$ and $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

ground

- $ground((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a collection $(n, \boldsymbol{p}_n)_{n \in N}$ such that $N \subseteq \mathcal{C}$ and $\boldsymbol{p}_n \in \mathcal{T}^k$ for some k where

  - $\mathbb{P}_2 = \mathbb{P}_1 \sqcup \mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$
    and $\mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in N} R_n[\boldsymbol{p}/\boldsymbol{p}_n]$ and
    - atom base $I(\mathbb{P}_1) \cup O(\mathbb{P}_1)$

    for $(R_c)_{c \in \mathcal{C}} = \boldsymbol{R}$
  - $V_2^t = \{a \in I(\mathbb{P}_2) \mid V_1(a) = t\}$
    $V_2^u = \{a \in I(\mathbb{P}_2) \mid V_1(a) = u\}$

ground

- $ground((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a collection $(n, \boldsymbol{p}_n)_{n \in N}$ such that $N \subseteq \mathcal{C}$ and $\boldsymbol{p}_n \in \mathcal{T}^k$ for some k where

  - $\mathbb{P}_2 = \mathbb{P}_1 \sqcup \mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$
    and $\mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in N} R_n[\boldsymbol{p}/\boldsymbol{p}_n]$ and
    - atom base $I(\mathbb{P}_1) \cup O(\mathbb{P}_1)$

    for $(R_c)_{c \in \mathcal{C}} = \boldsymbol{R}$
  - $V_2^t = \{a \in I(\mathbb{P}_2) \mid V_1(a) = t\}$
    $V_2^u = \{a \in I(\mathbb{P}_2) \mid V_1(a) = u\}$

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ while $ground((n, \boldsymbol{p}))(ground((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $head(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $\mathit{ground}((n,\boldsymbol{p}),(n,\boldsymbol{p}))(s) = \mathit{ground}((n,\boldsymbol{p}))(s)$ while $\mathit{ground}((n,\boldsymbol{p}))(\mathit{ground}((n,\boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $\mathit{head}(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $\mathit{ground}((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = \mathit{ground}((n, \boldsymbol{p}))(s)$ while $\mathit{ground}((n, \boldsymbol{p}))(\mathit{ground}((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $\mathit{head}(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n,\boldsymbol{p}),(n,\boldsymbol{p}))(s) = ground((n,\boldsymbol{p}))(s)$ while $ground((n,\boldsymbol{p}))(ground((n,\boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $head(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

assignExternal

- $assignExternal(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$
  for a ground atom a and $v \in \{t, u, f\}$ where
  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

Only input atoms, that is, non-overwritten externals are affected

assignExternal

- $\mathit{assignExternal}(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$

  for a ground atom a and $v \in \{t, u, f\}$ where
  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

- Note Only input atoms, that is, non-overwritten externals are affected

assignExternal

- $\mathit{assignExternal}(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$

  for a ground atom a and $v \in \{t, u, f\}$ where

  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

- Note Only input atoms, that is, non-overwritten externals are affected

releaseExternal

- *releaseExternal*$(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

releaseExternal only affects input atoms; defined atoms remain unaffected

A released atom can never be re-defined, neither by a rule nor an external declaration

A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and
    $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and
    $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and
    $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and
    $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

solve

- $solve((A^t, A^f)) : (\boldsymbol{R}, \mathbb{P}, V) \mapsto (\boldsymbol{R}, \mathbb{P}, V)$ prints the set

$$\{X \mid X \text{ is a stable model of } \mathbb{P} \text{ wrt } V \text{ st } A^t \subseteq X \text{ and } A^f \cap X = \emptyset\}$$

where the stable models of a module $\mathbb{P}$ wrt an assignment V are given by the stable models of the program

$$P(\mathbb{P}) \cup \{a \leftarrow \mid a \in V^t\} \cup \{\{a\} \leftarrow \mid a \in V^u\}$$

solve

- $solve((A^t, A^f)) : (\boldsymbol{R}, \mathbb{P}, V) \mapsto (\boldsymbol{R}, \mathbb{P}, V)$ prints the set

$$\{X \mid X \text{ is a stable model of } \mathbb{P} \text{ wrt } V \text{ st } A^t \subseteq X \text{ and } A^f \cap X = \emptyset\}$$

  where the stable models of a module $\mathbb{P}$ wrt an assignment V are given by the stable models of the program

$$P(\mathbb{P}) \cup \{a \leftarrow \mid a \in V^t\} \cup \{\{a\} \leftarrow \mid a \in V^u\}$$

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- main routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- main routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

`#script` declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

#script declaration

- A script declaration is of form

  #script(python) P #end

  where P is a Python program
- Analogously for Lua
- main routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

`#script` declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

`#script` declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Examples

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Extensible programs

- Initial *clingo* state

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow {\sim}p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow {\sim}p(n+1), {\sim}p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow {\sim}p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow {\sim}p(n+1), {\sim}p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object 'prg'

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external } p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external } p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external } p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external } p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object 'prg'

$$create(R) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

  where R is the list of rules and declarations in Line 1-8 and

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object ‘prg’

$$create(R) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset,\emptyset,\emptyset), (\emptyset,\emptyset))$$

where R is the list of rules and declarations in Line 1-8 and

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow {\sim}p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow {\sim}p(n+1), {\sim}p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$
- Note $create(R)$ is invoked implicitly to create *clingo* object ‘prg’

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
   #external p(1;2;3).
   p(0) :- p(3).
   p(0) :- not p(0).

   #program succ(n).
   #external p(n+3).
   p(n) :- p(n+3).
   p(n) :- not p(n+1), not p(n+2).

   #script(python)
   from clingo import Fun
   def main(prg):
>>     prg.ground([("base", [])])
       prg.assign_external(Fun("p", [3]), True)
       prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
       prg.solve()
   #end.
```

`prg.ground([("base", [])])`

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\mathtt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\begin{aligned}\mathbb{P}_1 &= \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\}) \\ &= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})\end{aligned}$$

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\mathtt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow \sim p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \sim p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\mathtt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

`prg.ground([("base", [])])`

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
>>      prg.ground([("base", [])])
        prg.assign_external(Fun("p", [3]), True)
        prg.solve()
        prg.assign_external(Fun("p", [3]), False)
        prg.solve()
        prg.ground([("succ", [1]),("succ", [2])])
        prg.solve()
        prg.ground([("succ", [3])])
        prg.solve()
#end.
```

Example

```
   #external p(1;2;3).
   p(0) :- p(3).
   p(0) :- not p(0).

   #program succ(n).
   #external p(n+3).
   p(n) :- p(n+3).
   p(n) :- not p(n+1), not p(n+2).

   #script(python)
   from clingo import Fun
   def main(prg):
       prg.ground([("base", [])])
>>     prg.assign_external(Fun("p", [3]), True)
       prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
       prg.solve()
   #end.
```

`prg.assign_external(Fun("p",[3]),True)`

- Global *clingo* state $(\boldsymbol{R}_1, \mathbb{P}_1, V_1)$
- Input assignment $p(3) \mapsto t$
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

`prg.assign_external(Fun("p",[3]),True)`

- Global *clingo* state $(\boldsymbol{R}_1, \mathbb{P}_1, V_1)$
- Input assignment $p(3) \mapsto t$
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
>>  prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
   #external p(1;2;3).
   p(0) :- p(3).
   p(0) :- not p(0).

   #program succ(n).
   #external p(n+3).
   p(n) :- p(n+3).
   p(n) :- not p(n+1), not p(n+2).

   #script(python)
   from clingo import Fun
   def main(prg):
       prg.ground([("base", [])])
       prg.assign_external(Fun("p", [3]), True)
>>     prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
       prg.solve()
   #end.
```

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

prg.solve()

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

prg.solve()

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

prg.solve()

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
>>  prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
>>  prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

```
prg.assign_external(Fun("p",[3]),False)
```

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input assignment $p(3) \mapsto f$
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

`prg.assign_external(Fun("p",[3]),False)`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input assignment $p(3) \mapsto f$
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
>>  prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
>>  prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state
$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$
- Print no stable model of $\mathbb{P}_3$ wrt V_3

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

- Print no stable model of $\mathbb{P}_3$ wrt V_3

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

- Print no stable model of $\mathbb{P}_3$ wrt V_3

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
>>  prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
>>  prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

prg.ground([("succ",[1]),("succ",[2])])

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$, including atom base
  $$I(\mathbb{P}_3) \cup O(\mathbb{P}_3) = \{p(0), p(1), p(2), p(3)\}$$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/1] \cup R(\mathtt{succ})[\mathtt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right) \quad \text{where}$$

$$P_4 = \left\{\begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

`prg.ground([("succ",[1]),("succ",[2])])`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$, including atom base
  $$I(\mathbb{P}_3) \cup O(\mathbb{P}_3) = \{p(0), p(1), p(2), p(3)\}$$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/1] \cup R(\mathtt{succ})[\mathtt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right) \quad \text{where}$$

$$P_4 = \left\{\begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

prg.ground([("succ",[1]),("succ",[2])])

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

prg.ground([("succ",[1]),("succ",[2])])

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix} \right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix} \right)$$

Potassco

`prg.ground([("succ",[1]),("succ",[2])])`

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

Potassco

prg.ground([("succ",[1]),("succ",[2])])

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}, \left\{\begin{matrix} p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(0), p(1), \\ p(2) \end{matrix}\right\}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{\begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right)$$

Potassco

`prg.ground([("succ",[1]),("succ",[2])])`

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

Potassco

prg.ground([("succ",[1]),("succ",[2])])

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{matrix}\right\}, \left\{\begin{matrix} p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(0), p(1), \\ p(2) \end{matrix}\right\}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{\begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{matrix}\right\}, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right)$$

Potassco

`prg.ground([("succ",[1]),("succ",[2])])`

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix} \right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix} \right)$$

`prg.ground([("succ",[1]),("succ",[2])])`

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{\begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}, \left\{\begin{array}{r} p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{r} p(0), p(1), \\ p(2) \end{array}\right\}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{\begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}, \left\{\begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{l} p(1), \\ p(2) \end{array}\right\}\right)$$

prg.ground([("succ",[1]),("succ",[2])])

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
>>  prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
>>  prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_4, V_3)$$

- Print no stable model of $\mathbb{P}_4$ wrt V_4

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_4, V_3)$$

- Print no stable model of $\mathbb{P}_4$ wrt V_4

prg.solve()

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_4, V_3)$$

- Print no stable model of $\mathbb{P}_4$ wrt V_4

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
>>  prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
>>  prg.ground([("succ", [3])])
    prg.solve()
#end.
```

`prg.ground([("succ", [3])])`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $$I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

prg.ground([("succ", [3])])

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $$I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

`prg.ground([("succ", [3])])`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $$I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ \ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

prg.ground([("succ", [3])])

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

where

$$\boldsymbol{R}_5 = (R(\texttt{base}), R(\texttt{succ}))$$

$$P(\mathbb{P}_5) = \left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4); \\ & p(3) \leftarrow p(6); & p(3) \leftarrow {\sim}p(4), {\sim}p(5) \end{array} \right\}$$

$$I(\mathbb{P}_5) = \{p(4), p(5), p(6)\}$$

$$O(\mathbb{P}_5) = \{p(0), p(1), p(2), p(3)\}$$

$$V_5 = (\emptyset, \emptyset)$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
>>  prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
   #external p(1;2;3).
   p(0) :- p(3).
   p(0) :- not p(0).

   #program succ(n).
   #external p(n+3).
   p(n) :- p(n+3).
   p(n) :- not p(n+1), not p(n+2).

   #script(python)
   from clingo import Fun
   def main(prg):
       prg.ground([("base", [])])
       prg.assign_external(Fun("p", [3]), True)
       prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
>>     prg.solve()
   #end.
```

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_5, \mathbb{P}_5, V_5)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_5, V_3)$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_5$ wrt V_5

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_5, \mathbb{P}_5, V_5)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_5, V_3)$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_5$ wrt V_5

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_5, \mathbb{P}_5, V_5)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_5, V_3)$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_5$ wrt V_5

simple.lp

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Clingo on the run

```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models       : 2+
Calls        : 4
Time         : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.010s
```

Clingo on the run

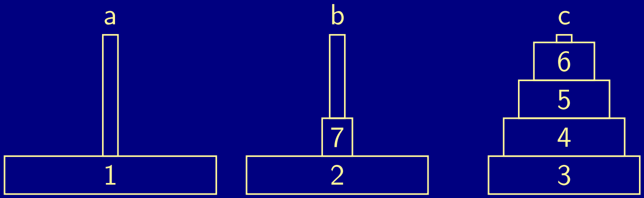
```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models       : 2+
Calls        : 4
Time         : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.010s
```

Potassco

Outline

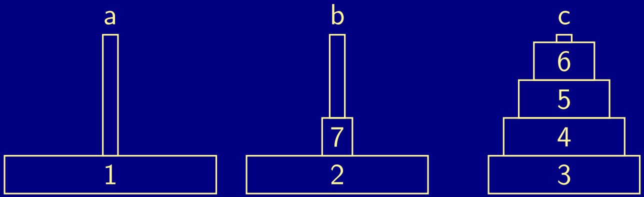
Towers of Hanoi Instance




```
peg(a;b;c).   disk(1..7).

    init_on(1,a).  init_on((2;7),b).  init_on((3;4;5;6),c).
goal_on((3;4),a).                   goal_on((1;2;5;6;7),c).
```

Towers of Hanoi Instance




```
peg(a;b;c).   disk(1..7).

    init_on(1,a).  init_on((2;7),b).  init_on((3;4;5;6),c).
goal_on((3;4),a).                   goal_on((1;2;5;6;7),c).
```

Towers of Hanoi Encoding

```
#program base.

on(D,P,0) :- init_on(D,P).
```

Towers of Hanoi Encoding

```
#program step(t).

1 { move(D,P,t) : disk(D), peg(P) } 1.

moved(D,t) :- move(D,_,t).
blocked(D,P,t) :- on(D+1,P,t-1), disk(D+1).
blocked(D,P,t) :- blocked(D+1,P,t), disk(D+1).
:- move(D,P,t), blocked(D-1,P,t).
:- moved(D,t), on(D,P,t-1), blocked(D,P,t).

on(D,P,t) :- on(D,P,t-1), not moved(D,t).
on(D,P,t) :- move(D,P,t).
:- not 1 { on(D,P,t) : peg(P) } 1, disk(D).
```

Towers of Hanoi Encoding

```
#program check(t).
#external query(t).

:- goal_on(D,P), not on(D,P,t), query(t).
```

Incremental Solving (ASP)

```
#script (python)

from clingo import SolveResult, Fun

def main(prg):
    ret, parts, step = SolveResult.UNSAT, [], 1
    parts.append(("base", []))
    while ret == SolveResult.UNSAT:
        parts.append(("step", [step]))
        parts.append(("check", [step]))
        prg.ground(parts)
        prg.release_external(Fun("query", [step-1]))
        prg.assign_external(Fun("query", [step]), True)
        ret, parts, step = prg.solve(), [], step+1

#end.
```

Incremental Solving (tohCtrl.lp)

```
#script (python)

from clingo import SolveResult, Fun

def main(prg):
    ret, parts, step = SolveResult.UNSAT, [], 1
    parts.append(("base", []))
    while ret == SolveResult.UNSAT:
        parts.append(("step", [step]))
        parts.append(("check", [step]))
        prg.ground(parts)
        prg.release_external(Fun("query", [step-1]))
        prg.assign_external(Fun("query", [step]), True)
        ret, parts, step = prg.solve(), [], step+1

#end.
```

Incremental Solving

```
$ clingo toh.lp tohCtrl.lp
clingo version 4.5.0
Reading from toh.lp ...
Solving...
Solving...
[...]
Solving...
Answer: 1
move(7,a,1)  move(6,b,2)  move(7,b,3)  move(5,a,4)  move(7,c,5)  move(6,a,6)  \
move(7,a,7)  move(4,b,8)  move(7,b,9)  move(6,c,10) move(7,c,11) move(5,b,12) \
move(1,c,13) move(7,a,14) move(6,b,15) move(7,b,16) move(3,a,17) move(7,c,18) \
move(6,a,19) move(7,a,20) move(5,c,21) move(7,b,22) move(6,c,23) move(7,c,24) \
move(4,a,25) move(7,a,26) move(6,b,27) move(7,b,28) move(5,a,29) move(7,c,30) \
move(6,a,31) move(7,a,32) move(2,c,33) move(7,c,34) move(6,b,35) move(7,b,36) \
move(5,c,37) move(7,a,38) move(6,c,39) move(7,c,40)
SATISFIABLE

Models       : 1+
Calls        : 40
Time         : 0.312s (Solving: 0.22s 1st Model: 0.01s Unsat: 0.21s)
CPU Time     : 0.300s
```

Incremental Solving

```
$ clingo toh.lp tohCtrl.lp
clingo version 4.5.0
Reading from toh.lp ...
Solving...
Solving...
[...]
Solving...
Answer: 1
move(7,a,1)  move(6,b,2)  move(7,b,3)  move(5,a,4)  move(7,c,5)  move(6,a,6)  \
move(7,a,7)  move(4,b,8)  move(7,b,9)  move(6,c,10) move(7,c,11) move(5,b,12) \
move(1,c,13) move(7,a,14) move(6,b,15) move(7,b,16) move(3,a,17) move(7,c,18) \
move(6,a,19) move(7,a,20) move(5,c,21) move(7,b,22) move(6,c,23) move(7,c,24) \
move(4,a,25) move(7,a,26) move(6,b,27) move(7,b,28) move(5,a,29) move(7,c,30) \
move(6,a,31) move(7,a,32) move(2,c,33) move(7,c,34) move(6,b,35) move(7,b,36) \
move(5,c,37) move(7,a,38) move(6,c,39) move(7,c,40)
SATISFIABLE

Models       : 1+
Calls        : 40
Time         : 0.312s (Solving: 0.22s 1st Model: 0.01s Unsat: 0.21s)
CPU Time     : 0.300s
```

Incremental Solving (Python)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (Python)

```
$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \
move(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3)  move(7,c,5)  \
move(7,a,1)  move(6,b,35) move(6,c,10) move(6,a,6)  move(6,b,2)  move(7,b,9)  \
move(7,a,7)  move(4,b,8)  move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \
move(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4)  \
move(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \
move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
```

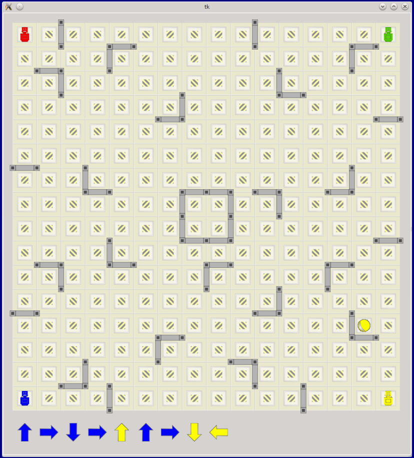
Incremental Solving (Python)

```
$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \
move(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3)  move(7,c,5)  \
move(7,a,1)  move(6,b,35) move(6,c,10) move(6,a,6)  move(6,b,2)  move(7,b,9)  \
move(7,a,7)  move(4,b,8)  move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \
move(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4)  \
move(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \
move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
```

Outline

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



- Four robots roaming
  - horizontally
  - vertically

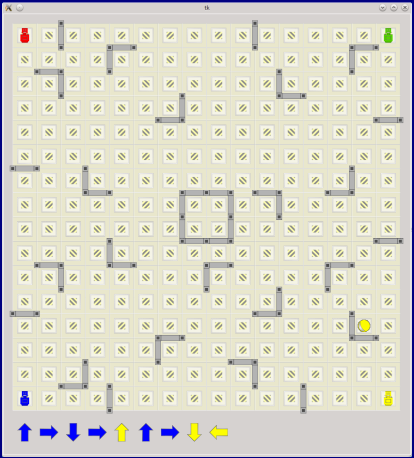
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target (sharing same color)

Potassco

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



- Four robots roaming
  - horizontally
  - vertically

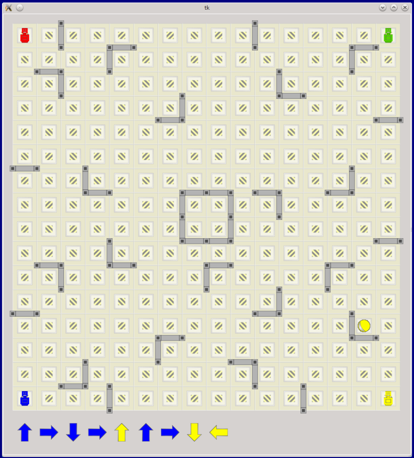
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target (sharing same color)

Potassco

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



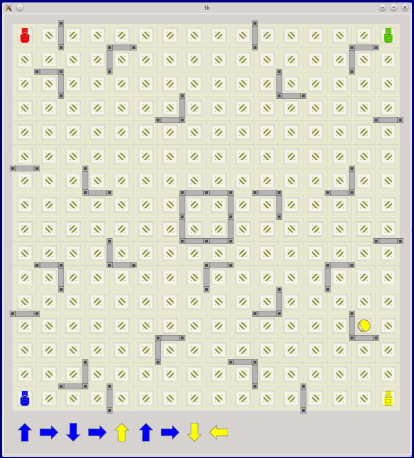
- Four robots
  roaming
  - horizontally
  - vertically

  up to blocking objects,
  ricocheting (optionally)

- Goal Robot on target
  (sharing same color)

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



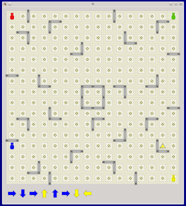
- Four robots
  roaming
  - horizontally
  - vertically

  up to blocking objects,
  ricocheting (optionally)

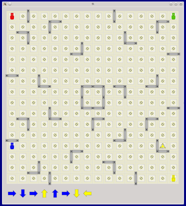
- Goal Robot on target
  (sharing same color)

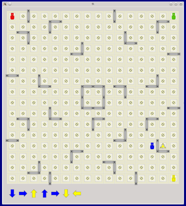
Potassco

Solving `goal(13)` from cornered robots (ctd)

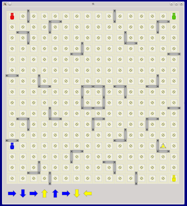


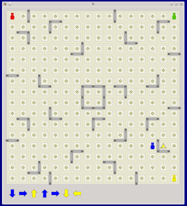
Solving `goal(13)` from cornered robots (ctd)

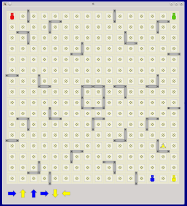




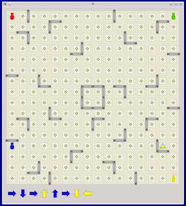
Solving `goal(13)` from cornered robots (ctd)

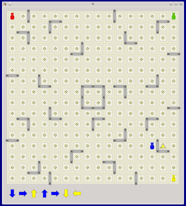


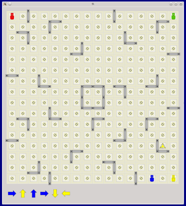


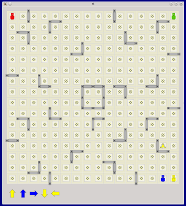


Solving `goal(13)` from cornered robots (ctd)

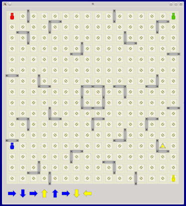
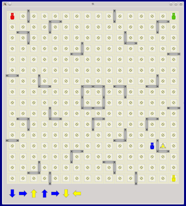
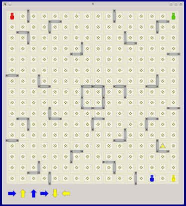
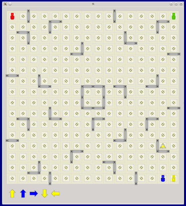
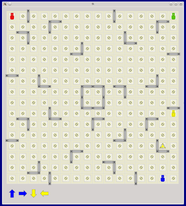




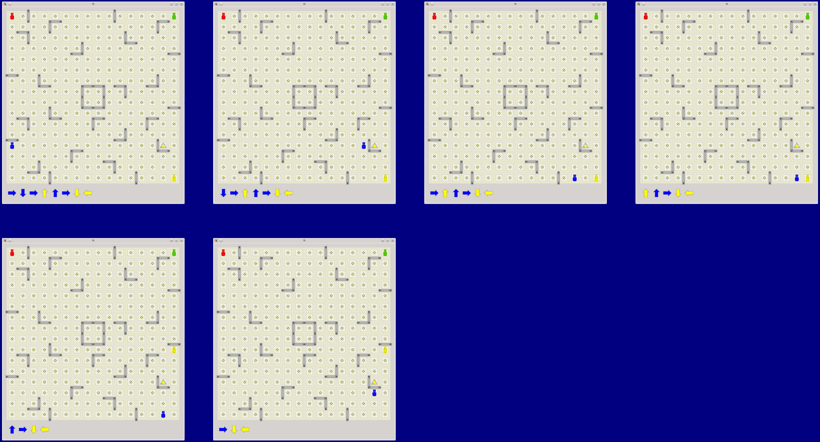




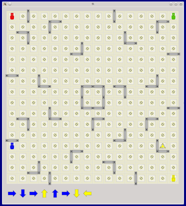
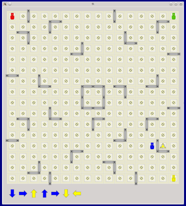
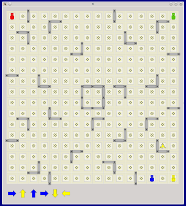
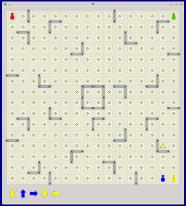
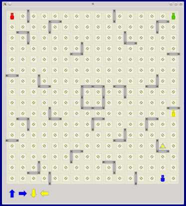
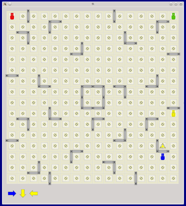
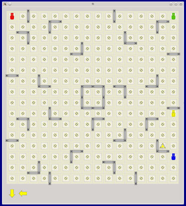
Solving `goal(13)` from cornered robots (ctd)

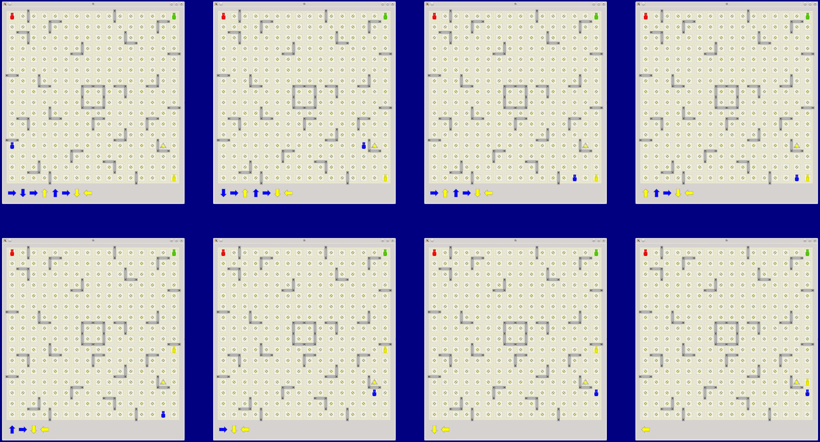
Solving `goal(13)` from cornered robots (ctd)



Solving `goal(13)` from cornered robots (ctd)

Solving `goal(13)` from cornered robots (ctd)



board.lp

```
dim(1..16).

barrier( 2, 1, 1, 0). barrier(13,11, 1, 0). barrier( 9, 7, 0, 1).
barrier(10, 1, 1, 0). barrier(11,12, 1, 0). barrier(11, 7, 0, 1).
barrier( 4, 2, 1, 0). barrier(14,13, 1, 0). barrier(14, 7, 0, 1).
barrier(14, 2, 1, 0). barrier( 6,14, 1, 0). barrier(16, 9, 0, 1).
barrier( 2, 3, 1, 0). barrier( 3,15, 1, 0). barrier( 2,10, 0, 1).
barrier(11, 3, 1, 0). barrier(10,15, 1, 0). barrier( 5,10, 0, 1).
barrier( 7, 4, 1, 0). barrier( 4,16, 1, 0). barrier( 8,10, 0,-1).
barrier( 3, 7, 1, 0). barrier(12,16, 1, 0). barrier( 9,10, 0,-1).
barrier(14, 7, 1, 0). barrier( 5, 1, 0, 1). barrier( 9,10, 0, 1).
barrier( 7, 8, 1, 0). barrier(15, 1, 0, 1). barrier(14,10, 0, 1).
barrier(10, 8,-1, 0). barrier( 2, 2, 0, 1). barrier( 1,12, 0, 1).
barrier(11, 8, 1, 0). barrier(12, 3, 0, 1). barrier(11,12, 0, 1).
barrier( 7, 9, 1, 0). barrier( 7, 4, 0, 1). barrier( 7,13, 0, 1).
barrier(10, 9,-1, 0). barrier(16, 4, 0, 1). barrier(15,13, 0, 1).
barrier( 4,10, 1, 0). barrier( 1, 6, 0, 1). barrier(10,14, 0, 1).
barrier( 2,11, 1, 0). barrier( 4, 7, 0, 1). barrier( 3,15, 0, 1).
barrier( 8,11, 1, 0). barrier( 8, 7, 0, 1).
```

targets.lp

```
#external goal(1..16).

target(red,   5, 2) :- goal(1).
target(red,  15, 2) :- goal(2).
target(green, 2, 3) :- goal(3).
target(blue, 12, 3) :- goal(4).
target(yellow, 7, 4) :- goal(5).
target(blue,  4, 7) :- goal(6).
target(green, 14, 7) :- goal(7).
target(yellow,11, 8) :- goal(8).
target(yellow, 5,10) :- goal(9).
target(green, 2,11) :- goal(10).
target(red,  14,11) :- goal(11).
target(green, 11,12) :- goal(12).
target(yellow,15,13) :- goal(13).
target(blue,  7,14) :- goal(14).
target(red,   3,15) :- goal(15).
target(blue, 10,15) :- goal(16).

robot(red;green;blue;yellow).
#external pos((red;green;blue;yellow),1..16,1..16).
```

ricochet.lp

```
time(1..horizon).
dir(-1,0;1,0;0,-1;0,1).

stop( DX, DY,X, Y   ) :- barrier(X,Y,DX,DY).
stop(-DX,-DY,X+DX,Y+DY) :- stop(DX,DY,X,Y).

pos(R,X,Y,0) :- pos(R,X,Y).

1 { move(R,DX,DY,T) : robot(R), dir(DX,DY) } 1 :- time(T).
move(R,T) :- move(R,_,_,T).

halt(DX,DY,X-DX,Y-DY,T) :- pos(_,X,Y,T), dir(DX,DY), dim(X-DX), dim(Y-DY),
                           not stop(-DX,-DY,X,Y), T < horizon.

goto(R,DX,DY,X,Y,T) :- pos(R,X,Y,T), dir(DX,DY), T < horizon.
goto(R,DX,DY,X+DX,Y+DY,T) :- goto(R,DX,DY,X,Y,T), dim(X+DX), dim(Y+DY),
                         not stop(DX,DY,X,Y), not halt(DX,DY,X,Y,T).

pos(R,X,Y,T) :- move(R,DX,DY,T), goto(R,DX,DY,X,Y,T-1),
              not goto(R,DX,DY,X+DX,Y+DY,T-1).
pos(R,X,Y,T) :- pos(R,X,Y,T-1), time(T), not move(R,T).

:- target(R,X,Y), not pos(R,X,Y,horizon).

#show move/4.
```

Potassco

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

optimization.lp

```
goon(T) :- target(R,X,Y), T = 0..horizon, not pos(R,X,Y,T).

:- move(R,DX,DY,T-1), time(T), not goon(T-1), not move(R,DX,DY,T).

#minimize{ 1,T : goon(T) }.
```

Solving goal(13) from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)   move(blue,1,0,2)    move(yellow,0,-1,3) move(blue,0,1,4)    move(yellow,-1,0,5) \
move(blue,1,0,6)    move(blue,0,-1,7)   move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10) \
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)
OPTIMUM FOUND

Models       : 12
  Optimum    : yes
Optimization : 9
Calls        : 1
Time         : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time     : 16.080s
```

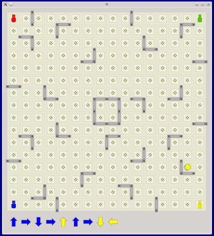
Solving `goal(13)` from cornered robots

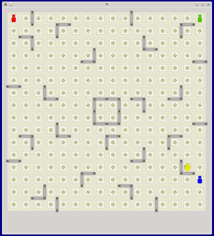
```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)   move(blue,1,0,2)    move(yellow,0,-1,3) move(blue,0,1,4)    move(yellow,-1,0,5) \
move(blue,1,0,6)    move(blue,0,-1,7)   move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10) \
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)
OPTIMUM FOUND

Models       : 12
  Optimum    : yes
Optimization : 9
Calls        : 1
Time         : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time     : 16.080s
```

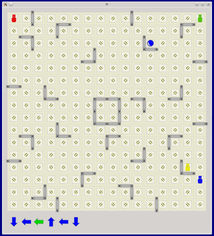
Playing in rounds

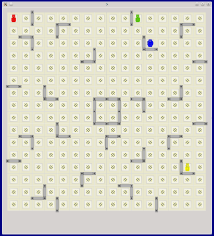
Round 1: `goal(13)`





Round 2: `goal(4)`





Control loop

1. Create an operational *clingo* object
2. Load and ground the logic programs encoding Ricochet Robot (relative to some fixed `horizon`) within the control object
3. While there is a goal, do the following
    1. Enforce the initial robot positions
    2. Enforce the current goal
    3. Solve the logic program contained in the control object

Ricochet Robot Player
ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Ricochet Robot Player
Setup and control loop

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Setup and control loop

```
horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal",  [4]),
             Fun("goal",  [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Setup and control loop

```
>> horizon   = 15
>> encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
>> positions = [Fun("pos", [Fun("red"),     1,  1]),
>>              Fun("pos", [Fun("blue"),    1, 16]),
>>              Fun("pos", [Fun("green"),  16,  1]),
>>              Fun("pos", [Fun("yellow"), 16, 16])]
>> sequence  = [Fun("goal", [13]),
>>              Fun("goal",  [4]),
>>              Fun("goal",  [7])]

   player = Player(horizon, positions, encodings)
   for goal in sequence:
       print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Setup and control loop

```
   horizon   = 15
   encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
   positions = [Fun("pos", [Fun("red"),     1,  1]),
                Fun("pos", [Fun("blue"),    1, 16]),
                Fun("pos", [Fun("green"),  16,  1]),
                Fun("pos", [Fun("yellow"), 16, 16])]
   sequence  = [Fun("goal", [13]),
                Fun("goal",  [4]),
                Fun("goal",  [7])]

>> player = Player(horizon, positions, encodings)
   for goal in sequence:
       print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Setup and control loop

```
   horizon   = 15
   encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
   positions = [Fun("pos", [Fun("red"),     1,  1]),
                Fun("pos", [Fun("blue"),    1, 16]),
                Fun("pos", [Fun("green"),  16,  1]),
                Fun("pos", [Fun("yellow"), 16, 16])]
   sequence  = [Fun("goal", [13]),
                Fun("goal",  [4]),
                Fun("goal",  [7])]

   player = Player(horizon, positions, encodings)
>> for goal in sequence:
>>     print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Setup and control loop

```
horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal",  [4]),
             Fun("goal",  [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Ricochet Robot Player

`__init__`

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

__init__

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
    def __init__(self, horizon, positions, files):
>>      self.last_positions = positions
>>      self.last_solution = None
>>      self.undo_external = []
>>      self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
>>  self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
>>      for x in files:
>>          self.ctl.load(x)
        self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
>>      self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

Ricochet Robot Player
solve

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

solve

```
    def solve(self, goal):
>>      for x in self.undo_external:
>>          self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model by passing user-defined `on_model` method

solve

```
    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
>>      self.undo_external = []
>>      for x in self.last_positions + [goal]:
>>          self.ctl.assign_external(x, True)
>>          self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

solve

```
    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
>>      self.last_solution = None
>>      self.ctl.solve(on_model=self.on_model)
>>      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Ricochet Robot Player

on_model

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
    def on_model(self, model):
>>        self.last_solution = model.atoms()
          self.last_positions = []
          for atom in model.atoms(Model.ATOMS):
              if (atom.name() == "pos" and
                      len(atom.args()) == 4 and
                      atom.args()[3] == self.horizon):
                  self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
    def on_model(self, model):
        self.last_solution = model.atoms()
>>      self.last_positions = []
>>      for atom in model.atoms(Model.ATOMS):
>>          if (atom.name() == "pos" and
>>                  len(atom.args()) == 4 and
>>                  atom.args()[3] == self.horizon):
>>              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
    def on_model(self, model):
            self.last_solution = model.atoms()
>>          self.last_positions = []
>>          for atom in model.atoms(Model.ATOMS):
>>              if (atom.name() == "pos" and
>>                      len(atom.args()) == 4 and
>>                      atom.args()[3] == self.horizon):
>>                  self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

Potassco

ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

[1] Y. Babovich and V. Lifschitz.
Computing answer sets using program completion.
Unpublished draft, 2003.

[2] C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.
Cambridge University Press, 2003.

[3] C. Baral, G. Brewka, and J. Schlipf, editors.
Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[4] C. Baral and M. Gelfond.
Logic programming and knowledge representation.
Journal of Logic Programming, 12:1–80, 1994.

[5] S. Baselice, P. Bonatti, and M. Gelfond.
Towards an integration of answer set and constraint solving.

In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

[6] A. Biere.
Adaptive restart strategies for conflict driven SAT solvers.
In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer-Verlag, 2008.

[7] A. Biere.
PicoSAT essentials.
Journal on Satisfiability, Boolean Modeling and Computation, 4:75–97, 2008.

[8] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*.

IOS Press, 2009.

[9] G. Brewka, T. Eiter, and M. Truszczyński.
Answer set programming at a glance.
Communications of the ACM, 54(12):92–103, 2011.

[10] G. Brewka, I. Niemelä, and M. Truszczyński.
Answer set optimization.
In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 867–872. Morgan Kaufmann Publishers, 2003.

[11] K. Clark.
Negation as failure.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[12] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.
Handbook of Tableau Methods.
Kluwer Academic Publishers, 1999.

[13] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov.
Complexity and expressive power of logic programming.
In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity (CCC'97)*, pages 82–101. IEEE Computer Society Press, 1997.

[14] M. Davis, G. Logemann, and D. Loveland.
A machine program for theorem-proving.
Communications of the ACM, 5:394–397, 1962.

[15] M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

[16] E. Di Rosa, E. Giunchiglia, and M. Maratea.
Solving satisfiability problems with preferences.
Constraints, 15(4):485–515, 2010.

[17] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub.

Conflict-driven disjunctive answer set solving.
In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008.

[18] C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub.
Heuristics in conflict resolution.
In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, number UNSW-CSE-TR-0819 in School of Computer Science and Engineering, The University of New South Wales, Technical Report Series, pages 141–149, 2008.

[19] N. Eén and N. Sörensson.
An extensible SAT-solver.
In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004.

[20] T. Eiter and G. Gottlob.
On the computational cost of disjunctive logic programming: Propositional case.
Annals of Mathematics and Artificial Intelligence, 15(3-4):289–323, 1995.

[21] T. Eiter, G. Ianni, and T. Krennwallner.
Answer Set Programming: A Primer.
In S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M. Rousset, and R. Schmidt, editors, *Fifth International Reasoning Web Summer School (RW'09)*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer-Verlag, 2009.

[22] F. Fages.
Consistency of Clark's completion and the existence of stable models.
Journal of Methods of Logic in Computer Science, 1:51–60, 1994.

[23] P. Ferraris.
Answer sets for propositional theories.

In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 119–131. Springer-Verlag, 2005.

[24] P. Ferraris and V. Lifschitz.
Mathematical foundations of answer set programming.
In S. Artëmov, H. Barringer, A. d'Avila Garcez, L. Lamb, and J. Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay*, volume 1, pages 615–664. College Publications, 2005.

[25] M. Fitting.
A Kripke-Kleene semantics for logic programs.
Journal of Logic Programming, 2(4):295–312, 1985.

[26] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub.
Abstract Gringo.
Theory and Practice of Logic Programming, 15(4-5):449–463, 2015.
Available at `http://arxiv.org/abs/1507.06576`.

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

[28] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.

[29] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
Engineering an incremental ASP solver.
In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2008.

[30] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.

On the implementation of weight constraint rules in conflict-driven ASP solvers.
In Hill and Warren [49], pages 250–264.

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

[32] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
clasp: A conflict-driven answer set solver.
In Baral et al. [3], pages 260–265.

[33] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. [3], pages 136–148.

[34] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso [74], pages 386–392.

[35] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Advanced preprocessing for answer set solving.
In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

[36] M. Gebser, B. Kaufmann, and T. Schaub.
The conflict-driven answer set solver clasp: Progress report.
In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 509–514. Springer-Verlag, 2009.

[37] M. Gebser, B. Kaufmann, and T. Schaub.
Solution enumeration for projected Boolean search problems.
In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*

(CPAIOR'09), volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.

[38] M. Gebser, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren [49], pages 235–249.

[39] M. Gebser and T. Schaub.
Tableau calculi for answer set programming.
In S. Etalle and M. Truszczyński, editors, *Proceedings of the Twenty-second International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 2006.

[40] M. Gebser and T. Schaub.
Generic tableaux for answer set programming.
In V. Dahl and I. Niemelä, editors, *Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2007.

[41] M. Gelfond.
Answer sets.
In V. Lifschitz, F. van Harmelen, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier Science, 2008.

[42] M. Gelfond and Y. Kahl.
Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.
Cambridge University Press, 2014.

[43] M. Gelfond and N. Leone.
Logic programming and knowledge representation — the A-Prolog perspective.
Artificial Intelligence, 138(1-2):3–38, 2002.

[44] M. Gelfond and V. Lifschitz.
The stable model semantics for logic programming.

In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[45] M. Gelfond and V. Lifschitz.
Logic programs with classical negation.
In D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, pages 579–597. MIT Press, 1990.

[46] E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.
Journal of Automated Reasoning, 36(4):345–377, 2006.

[47] K. Gödel.
Zum intuitionistischen Aussagenkalkül.
In *Anzeiger der Akademie der Wissenschaften in Wien*, page 65–66. 1932.

[48] A. Heyting.

Die formalen Regeln der intuitionistischen Logik.
In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, page 42–56. 1930.
Reprint in Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik, Akademie-Verlag, 1986.

[49] P. Hill and D. Warren, editors.
Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[50] J. Huang.
The effect of restarts on the efficiency of clause learning.
In Veloso [74], pages 2318–2323.

[51] K. Konczak, T. Linke, and T. Schaub.
Graphs and colorings for answer set programming.
Theory and Practice of Logic Programming, 6(1-2):61–106, 2006.

[52] J. Lee.

A model-theoretic counterpart of loop formulas.
In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

[53] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello.
The DLV system for knowledge representation and reasoning.
ACM Transactions on Computational Logic, 7(3):499–562, 2006.

[54] V. Lifschitz.
Answer set programming and plan generation.
Artificial Intelligence, 138(1-2):39–54, 2002.

[55] V. Lifschitz.
Introduction to answer set programming.
Unpublished draft, 2004.

[56] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?

Potassco

ACM Transactions on Computational Logic, 7(2):261–268, 2006.

[57] F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.
Artificial Intelligence, 157(1-2):115–137, 2004.

[58] V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.
Artifical Intelligence. Springer-Verlag, 1993.

[59] V. Marek and M. Truszczyński.
Stable models and an alternative logic programming paradigm.
In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[60] J. Marques-Silva, I. Lynce, and S. Malik.
Conflict-driven clause learning SAT solvers.
In Biere et al. [8], chapter 4, pages 131–153.

[61] J. Marques-Silva and K. Sakallah.

GRASP: A search algorithm for propositional satisfiability.
IEEE Transactions on Computers, 48(5):506–521, 1999.

[62] V. Mellarkod and M. Gelfond.
Integrating answer set reasoning with constraint solving techniques.
In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.

[63] V. Mellarkod, M. Gelfond, and Y. Zhang.
Integrating answer set programming and constraint logic programming.
Annals of Mathematics and Artificial Intelligence, 53(1-4):251–287, 2008.

[64] D. Mitchell.
A SAT solver primer.
Bulletin of the European Association for Theoretical Computer Science, 85:112–133, 2005.

[65] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

[66] I. Niemelä.
Logic programs with stable model semantics as a constraint programming paradigm.
Annals of Mathematics and Artificial Intelligence, 25(3-4):241–273, 1999.

[67] R. Nieuwenhuis, A. Oliveras, and C. Tinelli.
Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T).
Journal of the ACM, 53(6):937–977, 2006.

[68] K. Pipatsrisawat and A. Darwiche.
A lightweight component caching scheme for satisfiability solvers.

In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[69] L. Ryan.
Efficient algorithms for clause-learning SAT solvers.
Master's thesis, Simon Fraser University, 2004.

[70] P. Simons, I. Niemelä, and T. Soininen.
Extending and implementing the stable model semantics.
Artificial Intelligence, 138(1-2):181–234, 2002.

[71] T. Son and E. Pontelli.
Planning with preferences using logic programming.
Theory and Practice of Logic Programming, 6(5):559–608, 2006.

[72] T. Syrjänen.
Lparse 1.0 user's manual, 2001.

[73] A. Van Gelder, K. Ross, and J. Schlipf.

The well-founded semantics for general logic programs.
Journal of the ACM, 38(3):620–650, 1991.

[74] M. Veloso, editor.
Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI/MIT Press, 2007.

[75] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik.
Efficient conflict driven learning in a Boolean satisfiability solver.
In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. ACM Press, 2001.