Formal Methods in Answer Set Programming

Vladimir Lifschitz

University of Texas

October 20, 2024

Part 1:

Gentle Inroduction to Answer Set Programming

What Is Answer Set Programming?

ASP is a form of declarative programming, which means that it does not involve encoding algorithms.

A program in a declarative language is an encoding of the problem itself, not of an algorithm.

A declarative programming system finds a solution by the process of automated reasoning.

What Is Answer Set Programming?

ASP is a form of declarative programming, which means that it does not involve encoding algorithms.

A program in a declarative language is an encoding of the problem itself, not of an algorithm.

A declarative programming system finds a solution by the process of automated reasoning.

In ASP, a program is a set of rules, and solutions are found by systems called answer set solvers.

The solver CLINGO was created at the University of Potsdam: `https://potassco.org`.

Answer Set Solvers Calculate Stable Models

Program:

```
q :- not p.
```

Output of CLINGO:

```
Answer: 1
q
```

Answer Set Solvers Calculate Stable Models

Program:	Output of CLINGO:
q :- not p.	Answer: 1 q
q :- not p. r :- q.	Answer: 1 q r

Answer Set Solvers Calculate Stable Models

Program:	Output of CLINGO:
q :- not p.	Answer: 1 q
q :- not p. r :- q.	Answer: 1 q r
q :- not p. p :- not q.	Answer: 1 p Answer: 2 q

Answer Set Solvers Calculate Stable Models

Program:	Output of CLINGO:
q :- not p.	Answer: 1 q
q :- not p. r :- q.	Answer: 1 q r
q :- not p. p :- not q.	Answer: 1 p Answer: 2 q

Mathematical definition:
Michael Gelfond and V.L., 1988; Kit Fine, 1989.

Using Answer Set Solvers for Search

"We encode planning problems in such a way that stable models of the encodings correspond to valid sequences of actions" (Yannis Dimopoulos, Bernhard Nebel and Jana Koehler, 1997).

Using Answer Set Solvers for Search

"We encode planning problems in such a way that stable models of the encodings correspond to valid sequences of actions" (Yannis Dimopoulos, Bernhard Nebel and Jana Koehler, 1997).

- generating plans for the Reaction Control System of the Space Shuttle
- reconstructing evolutionary trees in biology and linguistics
- configuring railway safety systems and products in automotive industry
- finding answers to biomedical queries
- haplotype inference in genetics
- optimizing positions of valves in a water distribution system
- team building in the port of Gioia Tauro

A Puzzle

A farmer must transport a fox, a goose and a bag of beans from one bank of a river to the other using a boat which can only hold one item in addition to the farmer. The fox cannot be left alone with the goose, and the goose cannot be left alone with the beans. How can he do it?

A Puzzle

A farmer must transport a fox, a goose and a bag of beans from one bank of a river to the other using a boat which can only hold one item in addition to the farmer. The fox cannot be left alone with the goose, and the goose cannot be left alone with the beans. How can he do it?

Crossing with: goose nothing fox goose . . .

A Puzzle

A farmer must transport a fox, a goose and a bag of beans from one bank of a river to the other using a boat which can only hold one item in addition to the farmer. The fox cannot be left alone with the goose, and the goose cannot be left alone with the beans. How can he do it?

Crossing with:		goose		nothing		fox		goose		...		
Timeline:	0	$\rightarrow$	1	$\rightarrow$	2	$\rightarrow$	3	$\rightarrow$	4	...	h	

A Puzzle

A farmer must transport a fox, a goose and a bag of beans from one bank of a river to the other using a boat which can only hold one item in addition to the farmer. The fox cannot be left alone with the goose, and the goose cannot be left alone with the beans. How can he do it?

Crossing with:	goose	nothing	fox	goose	...
Timeline:	0 →	1 →	2 →	3 →	4 ... h
On the left bank:	boat fox goose beans	fox beans	boat fox beans	beans	boat goose beans

Solving the Puzzle

```
% definitions

item(fox).
item(goose).
item(beans).

load(I) :- item(I).
load(nothing).

opposite(left,right).
opposite(right,left).
```

Solving the Puzzle

```
% definitions

item(fox).
item(goose).
item(beans).

load(I) :- item(I).
load(nothing).

opposite(left,right).
opposite(right,left).

% possible actions

{cross(L,T)} :- load(L), 0 <= T < h.      % choice rule
:- cross(L1,T), cross(L2,T), L1 != L2.    % constraint
```

Solving the Puzzle, cont'd

```
% effects of actions

loc(boat,B1,T+1) :- cross(L,T), loc(boat,B,T), opposite(B,B1).
loc(I,B1,T+1) :- cross(I,T), item(I), loc(I,B,T), opposite(B,B1).
loc(I,B,T+1) :- cross(L,T), loc(I,B,T), item(I), I != L.
```

Solving the Puzzle, cont'd

```
% effects of actions

loc(boat,B1,T+1) :- cross(L,T), loc(boat,B,T), opposite(B,B1).
loc(I,B1,T+1) :- cross(I,T), item(I), loc(I,B,T), opposite(B,B1).
loc(I,B,T+1) :- cross(L,T), loc(I,B,T), item(I), I != L.

% initial conditions

loc(boat,left,0).
loc(I,left,0) :- item(I).
```

Solving the Puzzle, cont'd

```
% effects of actions

loc(boat,B1,T+1) :- cross(L,T), loc(boat,B,T), opposite(B,B1).
loc(I,B1,T+1) :- cross(I,T), item(I), loc(I,B,T), opposite(B,B1).
loc(I,B,T+1) :- cross(L,T), loc(I,B,T), item(I), I != L.

% initial conditions

loc(boat,left,0).
loc(I,left,0) :- item(I).

% prohibited states

:- loc(fox,B,T), loc(goose,B,T), not loc(boat,B,T).
:- loc(goose,B,T), loc(beans,B,T), not loc(boat,B,T).
```

Solving the Puzzle, cont'd

```
% goal

:- not loc(boat,right,h).
:- not loc(I,right,h), item(I).
```

Solving the Puzzle, cont'd

```
% goal

:- not loc(boat,right,h).
:- not loc(I,right,h), item(I).

% directives

#const h=7.
#show cross/2.
```

Solving the Puzzle, cont'd

```
% goal

:- not loc(boat,right,h).
:- not loc(I,right,h), item(I).

% directives

#const h=7.
#show cross/2.
```

Output of CLINGO:

```
Answer: 1
cross(goose,0) cross(nothing,1) cross(beans,2) cross(goose,3)
cross(fox,4) cross(nothing,5) cross(goose,6)
Answer: 2
cross(goose,0) cross(nothing,1) cross(fox,2) cross(goose,3)
cross(beans,4) cross(nothing,5) cross(goose,6)
```

Grounding and Safety

The process of calculating stable models begins with *grounding:* replacing the program with a program without variables that has the same stable models.

Grounding and Safety

The process of calculating stable models begins with *grounding:* replacing the program with a program without variables that has the same stable models.

Example: in the program

```
item(fox).   item(goose).   item(beans).   . . .
:- not loc(I,right,h), item(I).                    (⋆)
```

rule $(\star)$ can be replaced by

```
:- not loc(fox,right,h).
:- not loc(goose,right,h).
:- not loc(beans,right,h).
```

Grounding and Safety

The process of calculating stable models begins with *grounding:* replacing the program with a program without variables that has the same stable models.

Example: in the program

```
item(fox).   item(goose).   item(beans).   . . .
:- not loc(I,right,h), item(I).                    (⋆)
```

rule $(\star)$ can be replaced by

```
:- not loc(fox,right,h).
:- not loc(goose,right,h).
:- not loc(beans,right,h).
```

If we drop 'item(I)' from $(\star)$, CLINGO will display an error message:

```
'I' is unsafe
grounding stopped because of errors
```

Making a List of Prime Numbers

Problem: Find all primes between a and b.

Making a List of Prime Numbers

Problem: Find all primes between a and b. Assume that $a > 1$.

Making a List of Prime Numbers

Problem: Find all primes between a and b. Assume that $a > 1$.

```
prime(I) :- a <= I <= b, not composite(I).
```

Making a List of Prime Numbers

Problem: Find all primes between a and b. Assume that $a > 1$.

```
prime(I) :- a <= I <= b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Making a List of Prime Numbers

Problem: Find all primes between a and b. Assume that $a > 1$.

```
prime(I) :- a <= I <= b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

This program is correct, but unsafe.

Making a List of Prime Numbers

Problem: Find all primes between a and b. Assume that $a > 1$.

```
prime(I) :- a <= I <= b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

This program is correct, but unsafe. Replace the second rule by

```
composite(I*J) :- 2 <= I <= b, 2 <= J <= b.
```

Making a List of Prime Numbers

Problem: Find all primes between a and b. Assume that $a > 1$.

```
prime(I) :- a <= I <= b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

This program is correct, but unsafe. Replace the second rule by

```
composite(I*J) :- 2 <= I <= b, 2 <= J <= b.
```

With the directives

```
#show prime/1.
#const a = 10.
#const b = 20.
```

CLINGO will produce the output

```
Answer: 1
prime(11) prime(13) prime(17) prime(19)
```

Developing an ASP Program

Unsafe program Π_1 (using interval notation):

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Safe program Π_2:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b.
```

Developing an ASP Program

Unsafe program Π_1 (using interval notation):

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Safe program Π_2:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b.
```

More efficient program Π_3:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b/I.
```

Developing an ASP Program

Unsafe program Π_1 (using interval notation):

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Safe program Π_2:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b.
```

More efficient program Π_3:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b/I.
```

Two stages of the programming process:

- creating a straightforward encoding
- making it safe for grounding and efficient for search, possibly in several steps

Part 2:

Gentle Inroduction to the Use of Formal Methods in Answer Set Programming

Formal Methods

We would like to use mathematically rigorous methods to verify that changes made in a program in the process of making it safe and more efficient are indeed equivalent transformations.

Formal Methods

We would like to use mathematically rigorous methods to verify that changes made in a program in the process of making it safe and more efficient are indeed equivalent transformations.

Can we use such methods to verify the correctness of the original version of the program?

Formal Methods

We would like to use mathematically rigorous methods to verify that changes made in a program in the process of making it safe and more efficient are indeed equivalent transformations.

Can we use such methods to verify the correctness of the original version of the program? Yes, if the given specification is mathematically precise.

Formal Methods

We would like to use mathematically rigorous methods to verify that changes made in a program in the process of making it safe and more efficient are indeed equivalent transformations.

Can we use such methods to verify the correctness of the original version of the program? Yes, if the given specification is mathematically precise.

We need

- a definition of equivalence of programs
- computational methods for verifying equivalence

Formal Methods

We would like to use mathematically rigorous methods to verify that changes made in a program in the process of making it safe and more efficient are indeed equivalent transformations.

Can we use such methods to verify the correctness of the original version of the program? Yes, if the given specification is mathematically precise.

We need

- a definition of equivalence of programs
- computational methods for verifying equivalence

Another application: checking that independent ASP solutions to the same problem are equivalent.

Equivalence of Programs

Two programs are considered equivalent if they exhibit the same external behavior for every reasonable input.

"Reasonable input": values of a and b should be specified; they should be integers, and the value of a should be greater than 1.

"External behavior": the set of atoms in the stable model that contain *prime*/1.

Equivalence of Programs

Two programs are considered equivalent if they exhibit the same external behavior for every reasonable input.

"Reasonable input": values of a and b should be specified; they should be integers, and the value of a should be greater than 1.

"External behavior": the set of atoms in the stable model that contain *prime*/1.

This is made precise in the article by Fandinno, Hansen, Lierler, L. and Temple that defined *equivalence with respect to a user guide* (TPLP, 2023).

Equivalence of Programs

Two programs are considered equivalent if they exhibit the same external behavior for every reasonable input.

"Reasonable input": values of a and b should be specified; they should be integers, and the value of a should be greater than 1.

"External behavior": the set of atoms in the stable model that contain *prime*/1.

This is made precise in the article by Fandinno, Hansen, Lierler, L. and Temple that defined *equivalence with respect to a user guide* (TPLP, 2023).

The user guide for prime number programs:

input: a -> integer.	output: prime/1.
input: b -> integer.	assumption: a > 1.

Program Completion

Program completion (Keith Clark, 1978) is a syntactic transformation that converts logic programs into first-order theories. For many programs, stable models can be characterized in terms of program completion (François Fages, 1994).

Program Completion

Program completion (Keith Clark, 1978) is a syntactic transformation that converts logic programs into first-order theories. For many programs, stable models can be characterized in terms of program completion (François Fages, 1994).

Rules containing a predicate symbol p in the head can be viewed as sufficient conditions for p. The *completed definition* of p is the formula expressing that collectively these conditions are necessary.

Program Completion

Program completion (Keith Clark, 1978) is a syntactic transformation that converts logic programs into first-order theories. For many programs, stable models can be characterized in terms of program completion (François Fages, 1994).

Rules containing a predicate symbol p in the head can be viewed as sufficient conditions for p. The *completed definition* of p is the formula expressing that collectively these conditions are necessary.

Program:
```
p(a).
p(X) :- q(X,Y).
```

Completion: $\forall X(p(X) \leftrightarrow X = a \vee \exists Y\, q(X, Y))$.

Program Completion

Program completion (Keith Clark, 1978) is a syntactic transformation that converts logic programs into first-order theories. For many programs, stable models can be characterized in terms of program completion (François Fages, 1994).

Rules containing a predicate symbol p in the head can be viewed as sufficient conditions for p. The *completed definition* of p is the formula expressing that collectively these conditions are necessary.

Program:
```
p(a).
p(X) :- q(X,Y).
```

Completion: $\forall X(p(X) \leftrightarrow X = a \vee \exists Y\, q(X,Y))$.

Program Π_1:
```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Completion:

$$\forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)),$$
$$\forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1)).$$

Relating Program Equivalence to Completion

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

```
input: a -> integer.      output: prime/1.
input: b -> integer.      assumption: a > 1.
```

in terms of the completions of Π_1 and Π_2:

$$\begin{array}{c} \forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)), \\ \forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1)); \end{array} \tag{1}$$

$$\begin{array}{c} \forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)), \\ \forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge 2 \leq I \leq b \wedge 2 \leq J \leq b)). \end{array} \tag{2}$$

Relating Program Equivalence to Completion

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

```
input: a -> integer.      output: prime/1.
input: b -> integer.      assumption: a > 1.
```

in terms of the completions of Π_1 and Π_2:

$$\begin{array}{c} \forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)), \\ \forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1)); \end{array} \tag{1}$$

$$\begin{array}{c} \forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)), \\ \forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge 2 \leq I \leq b \wedge 2 \leq J \leq b)). \end{array} \tag{2}$$

First try: (1) is equivalent to (2) under the assumption $a > 1$. This is not satisfactory, because the definitions of *composite* in (1) and (2) are not equivalent.

Relating Program Equivalence to Completion

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

```
input: a -> integer.      output: prime/1.
input: b -> integer.      assumption: a > 1.
```

in terms of the completions of Π_1 and Π_2:

$$\begin{array}{c} \forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)), \\ \forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1)); \end{array} \tag{1}$$

$$\begin{array}{c} \forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)), \\ \forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge 2 \leq I \leq b \wedge 2 \leq J \leq b)). \end{array} \tag{2}$$

First try: (1) is equivalent to (2) under the assumption $a > 1$. This is not satisfactory, because the definitions of *composite* in (1) and (2) are not equivalent. We need a definition of completion that reflects the difference between output predicates and auxiliary predicates.

Forgetting

9 is the square of 3: $9 = 3^2$.

9 is a complete square: $\exists n(9 = n^2)$.

Forgetting

9 is the square of 3: $9 = 3^2$.
9 is a complete square: $\exists n(9 = n^2)$.

To forget, replace a constant by an existentially quantified variable.

Forgetting

9 is the square of 3: $9 = 3^2$.
9 is a complete square: $\exists n(9 = n^2)$.

To forget, replace a constant by an existentially quantified variable.

Completion of Π_1:

$$\forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg composite(I)) \wedge$$
$$\forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1)).$$

$Comp_1$, the completion of Π_1 with composite forgotten:

$$\exists C(\forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg C(I)) \wedge$$
$$\forall N(C(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1))).$$

Forgetting

9 is the square of 3: $9 = 3^2$.
9 is a complete square: $\exists n(9 = n^2)$.

To forget, replace a constant by an existentially quantified variable.

Completion of Π_1:

$$\forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg\, composite(I)) \wedge$$
$$\forall N(composite(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1)).$$

$Comp_1$, the completion of Π_1 with composite forgotten:

$$\exists C(\forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg\, C(I)) \wedge$$
$$\forall N(C(N) \leftrightarrow \exists IJ(N = I \times J \wedge I > 1 \wedge J > 1))).$$

$Comp_2$, the completion of Π_2 with composite forgotten:

$$\exists C(\forall I(prime(I) \leftrightarrow a \leq I \leq b \wedge \neg\, C(I)) \wedge$$
$$\forall N(C(N) \leftrightarrow \exists IJ(N = I \times J \wedge 2 \leq I \leq b \wedge 2 \leq J \leq b))).$$

Relating Program Equivalence to Completion, cont'd

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

input: a -> integer.	output: prime/1.
input: b -> integer.	assumption: a $>$ 1.

in terms of the completions of Π_1 and Π_2.

Relating Program Equivalence to Completion, cont'd

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

input: a -> integer.	output: prime/1.
input: b -> integer.	assumption: a $>$ 1.

in terms of the completions of Π_1 and Π_2.

Second try: Comp_1 is equivalent to Comp_2 assuming that $a > 1$.

Relating Program Equivalence to Completion, cont'd

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

input: a -> integer.	output: prime/1.
input: b -> integer.	assumption: $a > 1$.

in terms of the completions of Π_1 and Π_2.

Second try: Comp_1 is equivalent to Comp_2 assuming that $a > 1$. This is not satisfactory, because the definition of equivalence of formulas in logic refers to all possible ways to interpret

- object constants $(1, 2, a, b)$,
- function constants $(\times)$,
- predicate constants $(\leq, >, \mathit{prime})$.

Relating Program Equivalence to Completion, cont'd

We would like to express the claim that Π_1 and Π_2 are equivalent w.r.t. the user guide

input: a -> integer.	output: prime/1.
input: b -> integer.	assumption: $a > 1$.

in terms of the completions of Π_1 and Π_2.

Second try: Comp_1 is equivalent to Comp_2 assuming that $a > 1$. This is not satisfactory, because the definition of equivalence of formulas in logic refers to all possible ways to interpret

- object constants $(1, 2, a, b)$,
- function constants $(\times)$,
- predicate constants $(\leq, >, \mathit{prime})$.

We are interested in "standard interpretations": variables range over the integers, and $1, 2, \times, \leq, >$ are interpreted in the usual way.

Verifying Equivalence w.r.t. User Guide

Given a user guide and two programs,

- construct completion formulas Comp_1, Comp_2;
- derive the formula

$$\mathrm{Assumptions} \to (\mathrm{Comp}_1 \leftrightarrow \mathrm{Comp}_2) \qquad (\star)$$

from a set of axioms that hold for all standard interpretations.

Verifying Equivalence w.r.t. User Guide

Given a user guide and two programs,

- construct completion formulas Comp_1, Comp_2;
- derive the formula

$$\mathrm{Assumptions} \to (\mathrm{Comp}_1 \leftrightarrow \mathrm{Comp}_2) \qquad (\star)$$

from a set of axioms that hold for all standard interpretations.

ANTHEM (https://potassco.org/anthem/) is a proof assistant that

- given a user guide and two programs, generates formula $(\star)$, and
- calls the theorem prover VAMPIRE to derive $(\star)$ from an appropriate set of axioms.

Verifying Equivalence w.r.t. User Guide

Given a user guide and two programs,

- construct completion formulas Comp_1, Comp_2;
- derive the formula

$$\mathrm{Assumptions} \to (\mathrm{Comp}_1 \leftrightarrow \mathrm{Comp}_2) \qquad (\star)$$

from a set of axioms that hold for all standard interpretations.

ANTHEM (https://potassco.org/anthem/) is a proof assistant that

- given a user guide and two programs, generates formula $(\star)$, and
- calls the theorem prover VAMPIRE to derive $(\star)$ from an appropriate set of axioms.

Contributors:

Jorge Fandinno	Zach Hansen	Jan Heuer
Yuliya Lierler	Vladimir Lifschitz	Patrick Lühne
Torsten Schaub	Tobias Stolzmann	Nathan Temple

Operation of ANTHEM

Program 1:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Program 2:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b.
```

User guide:

```
input: a -> integer.
input: b -> integer.
output: prime/1.
assumption: a > 1.
```

Operation of ANTHEM

Program 1:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Program 2:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b.
```

User guide:

```
input: a -> integer.
input: b -> integer.
output: prime/1.
assumption: a > 1.
```

Output of ANTHEM

```
Success! Anthem found a proof of equivalence. (158355 ms)
```

Operation of ANTHEM

Program 1:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I > 1, J > 1.
```

Program 2:

```
prime(I) :- I = a..b, not composite(I).
composite(I*J) :- I = 2..b, J = 2..b.
```

User guide:

```
input: a -> integer.
input: b -> integer.
output: prime/1.
assumption: a > 1.
```

Output of ANTHEM

```
Success! Anthem found a proof of equivalence. (158355 ms)
```

Optional 4th argument: "proof outline".

Calculating Exact Covers

An *exact cover* of a collection S of sets is a subcollection S' of S such that each element of the union of all sets in S belongs to exactly one set in S'.

Example: $S = \{\{a\}, \{a, b\}, \{b, c\}\}$; $S' = \{\{a\}, \{b, c\}\}$.

Calculating Exact Covers

An *exact cover* of a collection S of sets is a subcollection S' of S such that each element of the union of all sets in S belongs to exactly one set in S'.

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$; $S' = \{\{a\},\{b,c\}\}$.

How do we encode S by a set of atoms?

$s(a,1),\ s(a,2),\ s(b,2),\ s(b,3),\ s(c,3);\ n = 3.$

Calculating Exact Covers

An *exact cover* of a collection S of sets is a subcollection S' of S such that each element of the union of all sets in S belongs to exactly one set in S'.

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$; $S' = \{\{a\},\{b,c\}\}$.

How do we encode S by a set of atoms?

$s(a,1),\ s(a,2),\ s(b,2),\ s(b,3),\ s(c,3);\ n = 3.$

How do we encode S' by a set of atoms?

$in\_cover(1),\ in\_cover(3).$

Calculating Exact Covers

An *exact cover* of a collection S of sets is a subcollection S' of S such that each element of the union of all sets in S belongs to exactly one set in S'.

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$; $S' = \{\{a\},\{b,c\}\}$.

How do we encode S by a set of atoms?

$s(a,1),\ s(a,2),\ s(b,2),\ s(b,3),\ s(c,3);\ n = 3.$

How do we encode S' by a set of atoms?

$in\_cover(1),\ in\_cover(3).$

Program:

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not _covered(X).
```

Calculating Exact Covers, cont'd

Program:

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not covered(X).
```

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$.

Calculating Exact Covers, cont'd

Program:

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not covered(X).
```

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$.

Input: `s(a,1). s(a,2). s(b,2). s(b,3). s(c,3). #const n=3.`

Calculating Exact Covers, cont'd

Program:

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not covered(X).
```

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$.

Input: `s(a,1). s(a,2). s(b,2). s(b,3). s(c,3). #const n=3.`

Output of CLINGO, with the directive `#show in_cover/1`:

```
in_cover(1) in_cover(3)
```

Calculating Exact Covers, cont'd

Program:

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not covered(X).
```

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$.

Input: `s(a,1). s(a,2). s(b,2). s(b,3). s(c,3). #const n=3.`

Output of CLINGO, with the directive `#show in_cover/1`:

```
in_cover(1) in_cover(3)
```

What user guide would we specify to reason about this program?

Calculating Exact Covers, cont'd

Program:

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not covered(X).
```

Example: $S = \{\{a\},\{a,b\},\{b,c\}\}$.

Input: `s(a,1). s(a,2). s(b,2). s(b,3). s(c,3). #const n=3.`

Output of CLINGO, with the directive `#show in_cover/1`:

`in_cover(1) in_cover(3)`

What user guide would we specify to reason about this program?

```
input: n -> integer.        assumption: n >= 0.
input: s/2.                 assumption: forall X I (s(X,I) -> 1 <= I <= n).
output: in_cover/1.
```

Conclusion

- To develop an ASP program, we improve a straightforward encoding to make it *safe* for grounding and *efficient* for search.

Conclusion

- To develop an ASP program, we improve a straightforward encoding to make it *safe* for grounding and *efficient* for search.
- Different versions of a program exhibit the same *external behavior* for every *reasonable input*.

Conclusion

- To develop an ASP program, we improve a straightforward encoding to make it *safe* for grounding and *efficient* for search.
- Different versions of a program exhibit the same *external behavior* for every *reasonable input*.
- This idea can be made precise by defining *external equivalence* as equivalence with respect to a *user guide*.

Conclusion

- To develop an ASP program, we improve a straightforward encoding to make it *safe* for grounding and *efficient* for search.
- Different versions of a program exhibit the same *external behavior* for every *reasonable input*.
- This idea can be made precise by defining *external equivalence* as equivalence with respect to a *user guide*.
- Under some conditions, external equivalence can be characterized in terms of *completed definitions*.

Conclusion

- To develop an ASP program, we improve a straightforward encoding to make it *safe* for grounding and *efficient* for search.
- Different versions of a program exhibit the same *external behavior* for every *reasonable input*.
- This idea can be made precise by defining *external equivalence* as equivalence with respect to a *user guide*.
- Under some conditions, external equivalence can be characterized in terms of *completed definitions*.
- The difference between output predicates and auxiliary predicates can be expressed using *second-order quantifiers*.

Conclusion

- To develop an ASP program, we improve a straightforward encoding to make it *safe* for grounding and *efficient* for search.
- Different versions of a program exhibit the same *external behavior* for every *reasonable input*.
- This idea can be made precise by defining *external equivalence* as equivalence with respect to a *user guide*.
- Under some conditions, external equivalence can be characterized in terms of *completed definitions*.
- The difference between output predicates and auxiliary predicates can be expressed using *second-order quantifiers*.
- In some cases, external equivalence of programs can be verified using the translator ANTHEM and the theorem prover VAMPIRE.