

# First-Order Theorem Proving and Vampire

Laura Kovács<sup>1,2</sup> and Martin Suda<sup>2</sup>

<sup>1</sup>TU Wien

<sup>2</sup>Chalmers

# Outline

Introduction

First-Order Logic and TPTP

Inference Systems

Saturation Algorithms

Redundancy Elimination

Equality

# General

The tool (VAMPIRE) is available at:

<http://forsyte.at/events/love2016/>

# First-Order Logic: Exercises

Which of the following statements are true?

# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;

# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;
2. First-order logic is **NP-complete**.

# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;
2. First-order logic is **NP-complete**.
3. First-order logic is **PSPACE-complete**.

# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;
2. First-order logic is **NP-complete**.
3. First-order logic is **PSPACE-complete**.
4. First-order logic is **decidable**.



# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;
2. First-order logic is **NP-complete**.
3. First-order logic is **PSPACE-complete**.
4. First-order logic is **decidable**.
5. In first-order logic you can use **quantifiers over sets**.

# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;
2. First-order logic is **NP-complete**.
3. First-order logic is **PSPACE-complete**.
4. First-order logic is **decidable**.
5. In first-order logic you can use **quantifiers over sets**.
6. One can **axiomatise integers** in first-order logic;

# First-Order Logic: Exercises

Which of the following statements are true?

1. First-order logic is an **extension of propositional logic**;
2. First-order logic is **NP-complete**.
3. First-order logic is **PSPACE-complete**.
4. First-order logic is **decidable**.
5. In first-order logic you can use **quantifiers over sets**.
6. One can **axiomatise integers** in first-order logic;
7. Having **proofs** is good.

# First-Order Theorem Proving. Example

**Group theory theorem:** if a group satisfies the identity  $x^2 = 1$ , then it is commutative.

# First-Order Theorem Proving. Example

**Group theory theorem:** if a group satisfies the identity  $x^2 = 1$ , then it is commutative.

**More formally:** in a group “**assuming** that  $x^2 = 1$  for all  $x$  **prove** that  $x \cdot y = y \cdot x$  holds for all  $x, y$ .”

# First-Order Theorem Proving. Example

**Group theory theorem:** if a group satisfies the identity  $x^2 = 1$ , then it is commutative.

**More formally:** in a group “**assuming** that  $x^2 = 1$  for all  $x$  **prove** that  $x \cdot y = y \cdot x$  holds for all  $x, y$ .”

**What is implicit:** axioms of the group theory.

$$\forall x(1 \cdot x = x)$$

$$\forall x(x^{-1} \cdot x = 1)$$

$$\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))$$

# Formulation in First-Order Logic

Axioms (of group theory):	$\forall x(1 \cdot x = x)$ $\forall x(x^{-1} \cdot x = 1)$ $\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))$
Assumptions:	$\forall x(x \cdot x = 1)$
Conjecture:	$\forall x \forall y(x \cdot y = y \cdot x)$

# In the TPTP Syntax

The **TPTP** library (**T**housands of **P**roblems for **T**heorem **P**rovers), <http://www.tptp.org> contains a large collection of first-order problems. For representing these problems it uses the **TPTP syntax**, which is understood by all modern theorem provers, including Vampire.



# In the TPTP Syntax

The **TPTP** library (**T**housands of **P**roblems for **T**heorem **P**rovers), <http://www.tptp.org> contains a large collection of first-order problems. For representing these problems it uses the **TPTP syntax**, which is understood by all modern theorem provers, including Vampire. In the TPTP syntax this group theory problem can be written down as follows:

```
%---- 1 * x = 1
fof(left_identity,axiom,
    ! [X] : mult(e,X) = X).
%---- i(x) * x = 1
fof(left_inverse,axiom,
    ! [X] : mult(inverse(X),X) = e).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,
    ! [X,Y,Z] : mult(mult(X,Y),Z) = mult(X,mult(Y,Z))).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
    ! [X] : mult(X,X) = e).
%---- prove x * y = y * x
fof(commutativity,conjecture,
    ! [X] : mult(X,Y) = mult(Y,X)).
```

# Running Vampire of a TPTP file

is easy: simply use

```
vampire <filename>
```

# Running Vampire of a TPTP file

is easy: simply use

```
vampire <filename>
```

One can also run Vampire with various options, some of them will be explained later. For example, save the group theory problem in a file `group.tptp` and try

```
vampire --thanks <your name> group.tptp
```

# Outline

Introduction

**First-Order Logic and TPTP**

Inference Systems

Saturation Algorithms

Redundancy Elimination

Equality

# First-Order Logic and TPTP

- ▶ **Language**: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.

# First-Order Logic and TPTP

- ▶ Language: variables, function and predicate (relation) symbols. A **constant symbol** is a special case of a function symbol.  
**In TPTP**: Variable names start with upper-case letters.

# First-Order Logic and TPTP

- ▶ Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.  
In TPTP: Variable names start with upper-case letters.
- ▶ **Terms**: variables, constants, and expressions  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms.

# First-Order Logic and TPTP

- ▶ Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.  
In TPTP: Variable names start with upper-case letters.
- ▶ Terms: variables, constants, and expressions  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms. Terms denote **domain elements**.



# First-Order Logic and TPTP

- ▶ Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.  
In TPTP: Variable names start with upper-case letters.
- ▶ Terms: variables, constants, and expressions  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms. Terms denote domain elements.
- ▶ **Atomic formula:** expression  $p(t_1, \dots, t_n)$ , where  $p$  is a predicate symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms.

# First-Order Logic and TPTP

- ▶ Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.  
In TPTP: Variable names start with upper-case letters.
- ▶ Terms: variables, constants, and expressions  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms. Terms denote domain elements.
- ▶ **Atomic formula:** expression  $p(t_1, \dots, t_n)$ , where  $p$  is a predicate symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms. Formulas denote **properties of domain elements**.
- ▶ All symbols are uninterpreted, apart from equality  $=$ .

# First-Order Logic and TPTP

- ▶ Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.  
In TPTP: Variable names start with upper-case letters.
- ▶ Terms: variables, constants, and expressions  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms. Terms denote domain elements.
- ▶ **Atomic formula:** expression  $p(t_1, \dots, t_n)$ , where  $p$  is a predicate symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms. Formulas denote properties of domain elements.
- ▶ All symbols are uninterpreted, apart from equality  $=$ .

FOL	TPTP
$\perp, \top$	<code>\$false, \$true</code>
$\neg a$	<code>~a</code>
$a_1 \wedge \dots \wedge a_n$	<code>a1 &amp; ... &amp; an</code>
$a_1 \vee \dots \vee a_n$	<code>a1   ...   an</code>
$a_1 \rightarrow a_2$	<code>a1 =&gt; a2</code>
$(\forall x_1) \dots (\forall x_n) a$	<code>! [X1, ..., Xn] : a</code>
$(\exists x_1) \dots (\exists x_n) a$	<code>? [X1, ..., Xn] : a</code>

## More on the TPTP Syntax

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
    mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

# More on the TPTP Syntax

## ► Comments;

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
    mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

## More on the TPTP Syntax

- ▶ Comments;
- ▶ Input formula names;

```
%---- 1 * x = x
fof(left_identity, axiom, (
  ! [X] : mult(e, X) = X )).
%---- i(x) * x = 1
fof(left_inverse, axiom, (
  ! [X] : mult(inverse(X), X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity, axiom, (
  ! [X, Y, Z] :
    mult(mult(X, Y), Z) = mult(X, mult(Y, Z)) )).
%---- x * x = 1
fof(group_of_order_2, hypothesis,
  ! [X] : mult(X, X) = e ).
%---- prove x * y = y * x
fof(commutativity, conjecture,
  ! [X, Y] : mult(X, Y) = mult(Y, X) ).
```

# More on the TPTP Syntax

- ▶ **Comments**;
- ▶ **Input formula names**;
- ▶ **Input formula roles** (very important);

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
    mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

# More on the TPTP Syntax

- ▶ Comments;
- ▶ Input formula names;
- ▶ Input formula roles (very important);
- ▶ Equality

```
%---- 1 * x = x
fof(left_identity, axiom, (
  ! [X] : mult(e, X) = X )).
%---- i(x) * x = 1
fof(left_inverse, axiom, (
  ! [X] : mult(inverse(X), X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity, axiom, (
  ! [X, Y, Z] :
    mult(mult(X, Y), Z) = mult(X, mult(Y, Z)) )).
%---- x * x = 1
fof(group_of_order_2, hypothesis,
  ! [X] : mult(X, X) = e ).
%---- prove x * y = y * x
fof(commutativity, conjecture,
  ! [X, Y] : mult(X, Y) = mult(Y, X) ).
```



# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

► Each inference derives a formula from zero or more other formulas;

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ **Input**, preprocessing, new symbols introduction, superposition calculus

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ Input, preprocessing, new symbols introduction, superposition calculus

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ Input, preprocessing, **new symbols introduction**, superposition calculus

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult(sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ Input, preprocessing, new symbols introduction, **superposition calculus**

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ Input, preprocessing, new symbols introduction, superposition calculus
- ▶ **Proof by refutation**, generating and simplifying inferences, unused formulas ...

# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ Input, preprocessing, new symbols introduction, superposition calculus
- ▶ Proof by refutation, **generating** and **simplifying** inferences, unused formulas ...



# Proof by Vampire (Slightly Modified)

Refutation found. Thanks to Tanya!

```
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sk0,sk1) != mult(sk1,sk0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sk0,sk1) != mult(sk1,sk0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sk0,sk1) != mult(sk1,sk0)
                                     [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0) [input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- ▶ Each inference derives a formula from zero or more other formulas;
- ▶ Input, preprocessing, new symbols introduction, superposition calculus
- ▶ Proof by refutation, generating and simplifying inferences, **unused formulas** ...

# Vampire

- ▶ **Completely automatic:** once you started a proof attempt, it can only be interrupted by terminating the process.

# Vampire

- ▶ **Completely automatic:** once you started a proof attempt, it can only be interrupted by terminating the process.
- ▶ **Champion** of the CASC world-cup in first-order theorem proving: won CASC 34 times.

# Our main applications

- ▶ Software and hardware verification;
- ▶ Static analysis of programs;
- ▶ Query answering in first-order knowledge bases (ontologies);
- ▶ Theorem proving in mathematics, especially in algebra;

# Our main applications

- ▶ Software and hardware verification;
- ▶ Static analysis of programs;
- ▶ Query answering in first-order knowledge bases (ontologies);
- ▶ Theorem proving in mathematics, especially in algebra;
- ▶ Writing papers and giving talks at various conferences and schools . . .

# What an Automatic Theorem Prover is Expected to Do

## Input:

- ▶ a set of **axioms** (first order formulas) or clauses;
- ▶ a **conjecture** (first-order formula or set of clauses).

## Output:

- ▶ **proof** (hopefully).

# Proof by Refutation

Given a problem with axioms and assumptions  $F_1, \dots, F_n$  and conjecture  $G$ ,

1. negate the conjecture;
2. establish **unsatisfiability** of the set of formulas  $F_1, \dots, F_n, \neg G$ .

# Proof by Refutation

Given a problem with axioms and assumptions  $F_1, \dots, F_n$  and conjecture  $G$ ,

1. negate the conjecture;
2. establish **unsatisfiability** of the set of formulas  $F_1, \dots, F_n, \neg G$ .

Thus, we reduce the theorem proving problem to the problem of **checking unsatisfiability**.



# Proof by Refutation

Given a problem with axioms and assumptions  $F_1, \dots, F_n$  and conjecture  $G$ ,

1. negate the conjecture;
2. establish **unsatisfiability** of the set of formulas  $F_1, \dots, F_n, \neg G$ .

Thus, we reduce the theorem proving problem to the problem of **checking unsatisfiability**.

In this formulation the negation of the conjecture  $\neg G$  is treated like any other formula. In fact, Vampire (and other provers) **internally treat conjectures differently, to make proof search more goal-oriented**.

# General Scheme (simplified)

- ▶ Read a problem;
- ▶ Determine proof-search options to be used for this problem;
- ▶ Preprocess the problem;
- ▶ Convert it into CNF;
- ▶ Run a saturation algorithm on it, try to derive  $\perp$ .
- ▶ If  $\perp$  is derived, report the result, maybe including a refutation.

# General Scheme (simplified)

- ▶ Read a problem;
- ▶ Determine proof-search options to be used for this problem;
- ▶ Preprocess the problem;
- ▶ Convert it into CNF;
- ▶ Run a **saturation algorithm** on it, try to derive  $\perp$ .
- ▶ If  $\perp$  is derived, report the result, maybe including a refutation.

Trying to derive  $\perp$  using a saturation algorithm is the **hardest part**, which in practice may not terminate or run out of memory.

# Outline

Introduction

First-Order Logic and TPTP

**Inference Systems**

Saturation Algorithms

Redundancy Elimination

Equality

# Inference System

- ▶ **inference** has the form

$$\frac{F_1 \quad \dots \quad F_n}{G},$$

where  $n \geq 0$  and  $F_1, \dots, F_n, G$  are formulas.

- ▶ The formula  $G$  is called the **conclusion** of the inference;
- ▶ The formulas  $F_1, \dots, F_n$  are called its **premises**.
- ▶ An **inference rule**  $R$  is a set of inferences.
- ▶ An **Inference system**  $\mathbb{I}$  is a set of inference rules.
- ▶ **Axiom**: inference rule with no premises.

# Derivation, Proof

- ▶ **Derivation** in an inference system  $\mathbb{I}$ : a tree built from inferences in  $\mathbb{I}$ .
- ▶ If the root of this derivation is  $E$ , then we say it is a **derivation of  $E$** .
- ▶ **Proof** of  $E$ : a finite derivation whose leaves are axioms.
- ▶ **Derivation of  $E$  from  $E_1, \dots, E_m$** : a finite derivation of  $E$  whose every leaf is either an axiom or one of the expressions  $E_1, \dots, E_m$ .

# Arbitrary First-Order Formulas

- ▶ A **first-order signature (vocabulary)**: function symbols (including constants), predicate symbols. **Equality** is part of the language.
- ▶ A set of **variables**.
- ▶ **Terms** are built using variables and function symbols. For example,  $f(x) + g(x)$ .
- ▶ **Atoms**, or **atomic formulas** are obtained by applying a predicate symbol to a sequence of terms. For example,  $p(a, x)$  or  $f(x) + g(x) \geq 2$ .
- ▶ **Formulas**: built from atoms using logical connectives  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\rightarrow$ ,  $\leftrightarrow$  and quantifiers  $\forall$ ,  $\exists$ . For example,  $(\forall x)x = 0 \vee (\exists y)y > x$ .

# Clauses

- ▶ **Literal:** either an atom  $A$  or its negation  $\neg A$ .
- ▶ **Clause:** a disjunction  $L_1 \vee \dots \vee L_n$  of literals, where  $n \geq 0$ .



# Clauses

- ▶ **Literal:** either an atom  $A$  or its negation  $\neg A$ .
- ▶ **Clause:** a disjunction  $L_1 \vee \dots \vee L_n$  of literals, where  $n \geq 0$ .
- ▶ **Empty clause**, denoted by  $\square$ : clause with 0 literals, that is, when  $n = 0$ .

# Clauses

- ▶ **Literal:** either an atom  $A$  or its negation  $\neg A$ .
- ▶ **Clause:** a disjunction  $L_1 \vee \dots \vee L_n$  of literals, where  $n \geq 0$ .
- ▶ **Empty clause**, denoted by  $\square$ : clause with 0 literals, that is, when  $n = 0$ .
- ▶ A formula in **Clausal Normal Form (CNF)**: a conjunction of clauses.

# Clauses

- ▶ **Literal**: either an atom  $A$  or its negation  $\neg A$ .
- ▶ **Clause**: a disjunction  $L_1 \vee \dots \vee L_n$  of literals, where  $n \geq 0$ .
- ▶ **Empty clause**, denoted by  $\square$ : clause with 0 literals, that is, when  $n = 0$ .
- ▶ A formula in **Clausal Normal Form (CNF)**: a conjunction of clauses.
- ▶ A clause is **ground** if it contains no variables.
- ▶ If a clause contains variables, we assume that it **implicitly universally quantified**. That is, we treat  $p(x) \vee q(x)$  as  $\forall x(p(x) \vee q(x))$ .

# Binary Resolution Inference System

The **binary resolution inference system**, denoted by **BR** is an inference system on **propositional** clauses (or **ground** clauses). It consists of two inference rules:

- ▶ **Binary resolution**, denoted by **BR**:

$$\frac{p \vee C_1 \quad \neg p \vee C_2}{C_1 \vee C_2} \text{ (BR).}$$

- ▶ **Factoring**, denoted by **Fact**:

$$\frac{L \vee L \vee C}{L \vee C} \text{ (Fact).}$$

# Soundness

- ▶ **An inference is sound** if the conclusion of this inference is a logical consequence of its premises.
- ▶ **An inference system is sound** if every inference rule in this system is sound.

# Soundness

- ▶ **An inference is sound** if the conclusion of this inference is a logical consequence of its premises.
- ▶ **An inference system is sound** if every inference rule in this system is sound.

$\mathbb{BR}$  is sound.

Consequence of soundness: let  $S$  be a set of clauses. If  $\square$  can be derived from  $S$  in  $\mathbb{BR}$ , then  $S$  is **unsatisfiable**.

## Example

Consider the following set of clauses

$$\{\neg p \vee \neg q, \neg p \vee q, p \vee \neg q, p \vee q\}.$$

The following derivation derives the empty clause from this set:

$$\frac{\frac{\frac{p \vee q \quad p \vee \neg q}{p \vee p} \text{ (BR)}}{p} \text{ (Fact)}}{\frac{\frac{\frac{\neg p \vee q \quad \neg p \vee \neg q}{\neg p \vee \neg p} \text{ (BR)}}{\neg p} \text{ (Fact)}}{\neg p} \text{ (BR)}}{\square}$$

Hence, this set of clauses is **unsatisfiable**.

# Can this be used for checking (un)satisfiability

1. What happens when the empty clause **cannot be derived** from  $S$ ?
2. **How** can one search for possible derivations of the empty clause?



# Can this be used for checking (un)satisfiability

## 1. Completeness.

*Let  $S$  be an unsatisfiable set of clauses. Then there exists a derivation of  $\square$  from  $S$  in  $\text{BR}$ .*

# Can this be used for checking (un)satisfiability

## 1. Completeness.

*Let  $S$  be an unsatisfiable set of clauses. Then there exists a derivation of  $\square$  from  $S$  in  $\text{BR}$ .*

## 2. We have to formalize search for derivations.

However, before doing this we will introduce a slightly more refined inference system.

# Selection Function

A **literal selection function** selects literals in a clause.

- ▶ If  $C$  is non-empty, then **at least one literal is selected** in  $C$ .

# Selection Function

A **literal selection function** selects literals in a clause.

- ▶ If  $C$  is non-empty, then **at least one literal is selected** in  $C$ .

We denote selected literals by underlining them, e.g.,

$$\underline{p} \vee \neg q$$

# Binary Resolution with Selection

We introduce a family of inference systems, parametrised by a literal selection function  $\sigma$ .

The **binary resolution inference system**, denoted by  $\text{BR}_\sigma$ , consists of two inference rules:

- ▶ **Binary resolution**, denoted by **BR**

$$\frac{p \vee C_1 \quad \neg p \vee C_2}{C_1 \vee C_2} \text{ (BR)}.$$

# Binary Resolution with Selection

We introduce a family of inference systems, parametrised by a literal selection function  $\sigma$ .

The **binary resolution inference system**, denoted by  $\mathbb{BR}_\sigma$ , consists of two inference rules:

- ▶ **Binary resolution**, denoted by **BR**

$$\frac{p \vee C_1 \quad \neg p \vee C_2}{C_1 \vee C_2} \text{ (BR).}$$

- ▶ **Positive factoring**, denoted by **Fact**:

$$\frac{p \vee \underline{p} \vee C}{p \vee C} \text{ (Fact).}$$

# Completeness?

Binary resolution with selection may be **incomplete**, even when factoring is unrestricted (also applied to negative literals).

# Completeness?

Binary resolution with selection may be **incomplete**, even when factoring is unrestricted (also applied to negative literals).

Consider this set of clauses:

- (1)  $\neg q \vee \underline{r}$
- (2)  $\neg p \vee \underline{q}$
- (3)  $\neg r \vee \underline{\neg q}$
- (4)  $\neg q \vee \underline{\neg p}$
- (5)  $\neg p \vee \underline{\neg r}$
- (6)  $\neg r \vee \underline{p}$
- (7)  $r \vee q \vee \underline{p}$



# Completeness?

Binary resolution with selection may be **incomplete**, even when factoring is unrestricted (also applied to negative literals).

Consider this set of clauses:

- (1)  $\neg q \vee \underline{r}$
- (2)  $\neg p \vee \underline{q}$
- (3)  $\neg r \vee \underline{\neg q}$
- (4)  $\neg q \vee \underline{\neg p}$
- (5)  $\neg p \vee \underline{\neg r}$
- (6)  $\neg r \vee \underline{p}$
- (7)  $r \vee q \vee \underline{p}$

It is unsatisfiable:

- (8)  $q \vee p$  (6, 7)
- (9)  $q$  (2, 8)
- (10)  $r$  (1, 9)
- (11)  $\neg q$  (3, 10)
- (12)  $\square$  (9, 11)

Note the **linear representation of derivations** (used by Vampire and many other provers).

However, any inference with selection applied to this set of clauses give either a clause in this set, or a clause containing a clause in this set.

# Literal Orderings

Take any **well-founded ordering**  $\succ$  on atoms, that is, an ordering such that there is no infinite decreasing chain of atoms:

$$A_0 \succ A_1 \succ A_2 \succ \dots$$

In the sequel  $\succ$  will always denote a well-founded ordering.

# Literal Orderings

Take any **well-founded ordering**  $\succ$  on atoms, that is, an ordering such that there is no infinite decreasing chain of atoms:

$$A_0 \succ A_1 \succ A_2 \succ \dots$$

In the sequel  $\succ$  will always denote a well-founded ordering.

Extend it to an ordering on literals by:

- ▶ If  $p \succ q$ , then  $p \succ \neg q$  and  $\neg p \succ q$ ;
- ▶  $\neg p \succ p$ .

# Orderings and Well-Behaved Selections

Fix an ordering  $\succ$ . A literal selection function is **well-behaved** if

- ▶ If all selected literals are **positive**, then **all maximal (w.r.t.  $\succ$ ) literals in  $C$  are selected**.

In other words, **either a negative literal is selected, or all maximal literals must be selected**.

# Orderings and Well-Behaved Selections

Fix an ordering  $\succ$ . A literal selection function is **well-behaved** if

- ▶ If all selected literals are **positive**, then **all maximal (w.r.t.  $\succ$ ) literals in  $C$  are selected**.

In other words, **either a negative literal is selected, or all maximal literals must be selected**.

To be well-behaved, we sometimes must select more than one different literal in a clause. Example:  $p \vee p$  or  $p(x) \vee p(y)$ .

# Completeness of Binary Resolution with Selection

Binary resolution with selection is **complete for every well-behaved selection function**.

# Completeness of Binary Resolution with Selection

Binary resolution with selection is **complete for every well-behaved selection function**.

Consider our previous example:

- (1)  $\neg q \vee \underline{r}$
- (2)  $\neg p \vee \underline{q}$
- (3)  $\neg r \vee \underline{\neg q}$
- (4)  $\neg q \vee \underline{\neg p}$
- (5)  $\neg p \vee \underline{\neg r}$
- (6)  $\neg r \vee \underline{p}$
- (7)  $r \vee q \vee \underline{p}$

A well-behaved selection function must satisfy:

- 1.  $r \succ q$ , because of (1)
- 2.  $q \succ p$ , because of (2)
- 3.  $p \succ r$ , because of (6)

There is no ordering that satisfies these conditions.

# Outline

Introduction

First-Order Logic and TPTP

Inference Systems

**Saturation Algorithms**

Redundancy Elimination

Equality



# How to Establish Unsatisfiability?

Completeness is formulated in terms of **derivability** of the empty clause  $\square$  from a set  $S_0$  of clauses in an inference system  $\mathbb{I}$ . However, this formulation gives **no hint on how to search** for such a derivation.

# How to Establish Unsatisfiability?

Completeness is formulated in terms of **derivability** of the empty clause  $\square$  from a set  $S_0$  of clauses in an inference system  $\mathbb{I}$ . However, this formulation gives **no hint on how to search** for such a derivation.

Idea:

- ▶ Take a set of clauses  $S$  (the **search space**), initially  $S = S_0$ . **Repeatedly apply inferences** in  $\mathbb{I}$  to clauses in  $S$  and add their conclusions to  $S$ , unless these conclusions are already in  $S$ .
- ▶ If, at any stage, we obtain  $\square$ , we terminate and **report unsatisfiability** of  $S_0$ .

# How to Establish Satisfiability?

When can we report **satisfiability**?

# How to Establish Satisfiability?

When can we report **satisfiability**?

When we build a set  $S$  such that any inference applied to clauses in  $S$  is already a member of  $S$ . Any such set of clauses is called **saturated** (with respect to  $\mathbb{I}$ ).

# How to Establish Satisfiability?

When can we report **satisfiability**?

When we build a set  $S$  such that any inference applied to clauses in  $S$  is already a member of  $S$ . Any such set of clauses is called **saturated** (with respect to  $\mathbb{I}$ ).

In first-order logic it is often the case that all saturated sets are infinite (due to undecidability), so in practice we can never build a saturated set.

The process of trying to build one is referred to as **saturation**.

# Saturated Set of Clauses

Let  $\mathbb{I}$  be an inference system on formulas and  $S$  be a set of formulas.

- ▶  $S$  is called **saturated with respect to  $\mathbb{I}$** , or simply  **$\mathbb{I}$ -saturated**, if for every inference of  $\mathbb{I}$  with premises in  $S$ , the conclusion of this inference also belongs to  $S$ .
- ▶ The **closure of  $S$  with respect to  $\mathbb{I}$** , or simply  **$\mathbb{I}$ -closure**, is the smallest set  $S'$  containing  $S$  and saturated with respect to  $\mathbb{I}$ .

# Inference Process

**Inference process:** sequence of sets of formulas  $S_0, S_1, \dots$ , denoted by

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$$

$(S_i \Rightarrow S_{i+1})$  is a **step** of this process.

# Inference Process

**Inference process:** sequence of sets of formulas  $S_0, S_1, \dots$ , denoted by

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$$

$(S_i \Rightarrow S_{i+1})$  is a **step** of this process.

We say that this step is an **I-step** if

1. there exists an inference

$$\frac{F_1 \quad \dots \quad F_n}{F}$$

in  $\mathbb{I}$  such that  $\{F_1, \dots, F_n\} \subseteq S_i$ ;

2.  $S_{i+1} = S_i \cup \{F\}$ .



# Inference Process

**Inference process:** sequence of sets of formulas  $S_0, S_1, \dots$ , denoted by

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$$

$(S_i \Rightarrow S_{i+1})$  is a **step** of this process.

We say that this step is an **I-step** if

1. there exists an inference

$$\frac{F_1 \quad \dots \quad F_n}{F}$$

in  $\mathbb{I}$  such that  $\{F_1, \dots, F_n\} \subseteq S_i$ ;

2.  $S_{i+1} = S_i \cup \{F\}$ .

An **I-inference process** is an inference process whose every step is an I-step.

# Property

Let  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  be an  $\mathbb{I}$ -inference process and a formula  $F$  belongs to some  $S_i$ . Then  $S_i$  is derivable in  $\mathbb{I}$  from  $S_0$ . In particular, every  $S_i$  is a subset of the  $\mathbb{I}$ -closure of  $S_0$ .

# Limit of a Process

The **limit** of an inference process  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  is the set of formulas  $\bigcup_i S_i$ .

# Limit of a Process

The **limit** of an inference process  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  is the set of formulas  $\bigcup_i S_i$ .

In other words, the limit is **the set of all derived formulas**.

# Limit of a Process

The **limit** of an inference process  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  is the set of formulas  $\bigcup_i S_i$ .

In other words, the limit is the set of all derived formulas.

Suppose that we have an infinite inference process such that  $S_0$  is **unsatisfiable** and we use a **sound and complete inference system**.

# Limit of a Process

The **limit** of an inference process  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  is the set of formulas  $\bigcup_i S_i$ .

In other words, the limit is the set of all derived formulas.

Suppose that we have an infinite inference process such that  $S_0$  is **unsatisfiable** and we use a **sound and complete inference system**.

**Question:** does completeness imply that the limit of the process contains the empty clause?

# Fairness

Let  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  be an inference process with the limit  $S_\infty$ .  
The process is called **fair** if for every  $\mathbb{I}$ -inference

$$\frac{F_1 \quad \dots \quad F_n}{F},$$

if  $\{F_1, \dots, F_n\} \subseteq S_\infty$ , then there exists  $i$  such that  $F \in S_i$ .

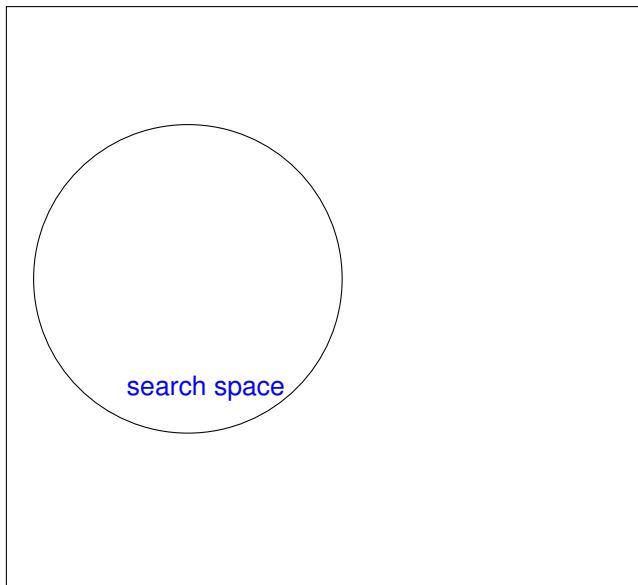
# Completeness, reformulated

**Theorem** Let  $\mathbb{I}$  be an inference system. The following conditions are equivalent.

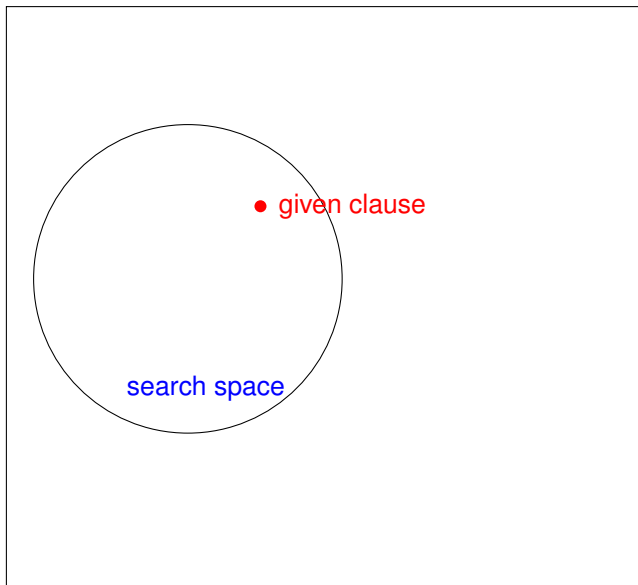
1.  $\mathbb{I}$  is complete.
2. For every unsatisfiable set of formulas  $S_0$  and any fair  $\mathbb{I}$ -inference process with the initial set  $S_0$ , the limit of this inference process contains  $\square$ .



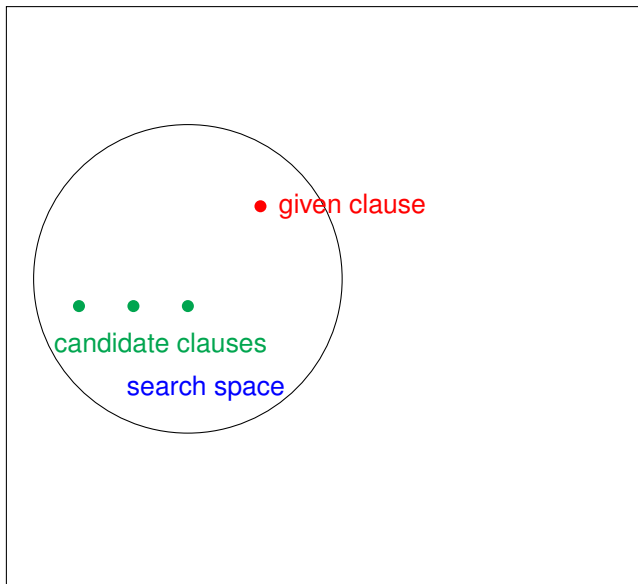
# Fair Saturation Algorithms: Inference Selection by Clause Selection



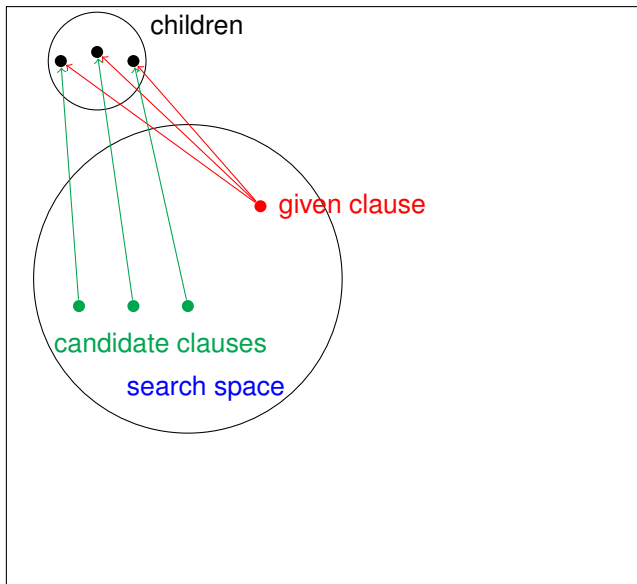
# Fair Saturation Algorithms: Inference Selection by Clause Selection



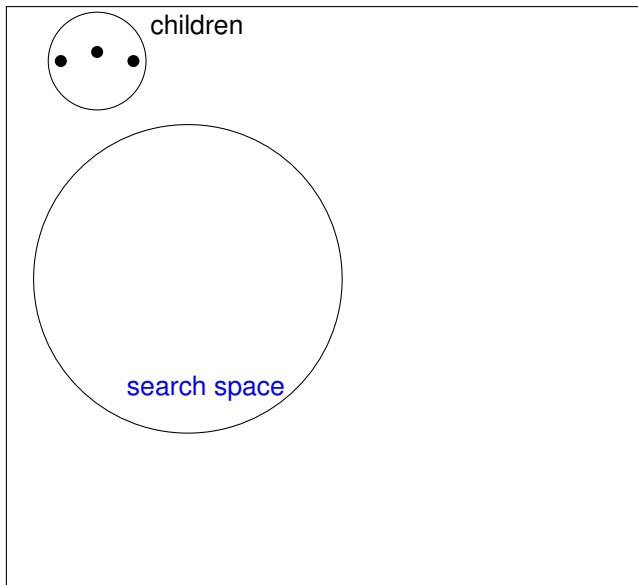
# Fair Saturation Algorithms: Inference Selection by Clause Selection



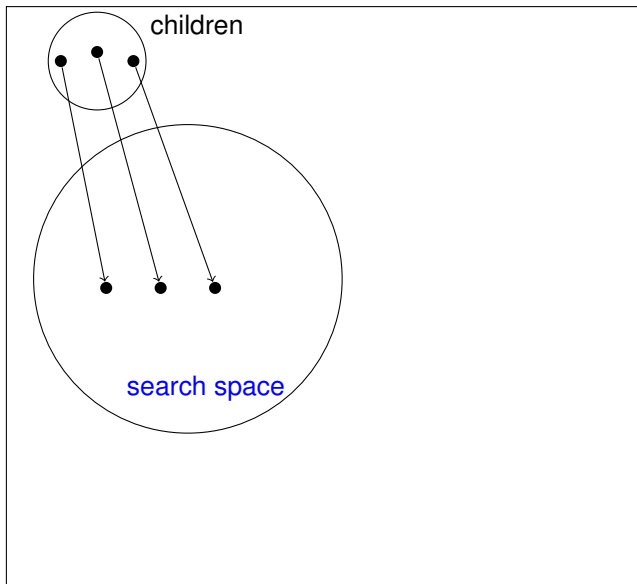
# Fair Saturation Algorithms: Inference Selection by Clause Selection



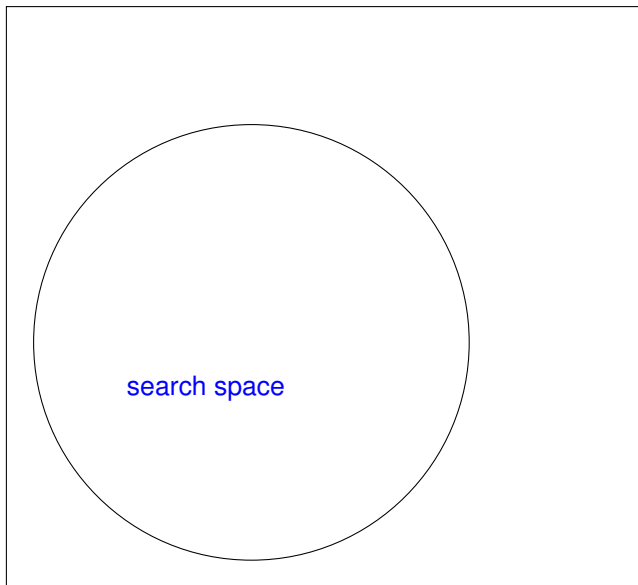
# Fair Saturation Algorithms: Inference Selection by Clause Selection



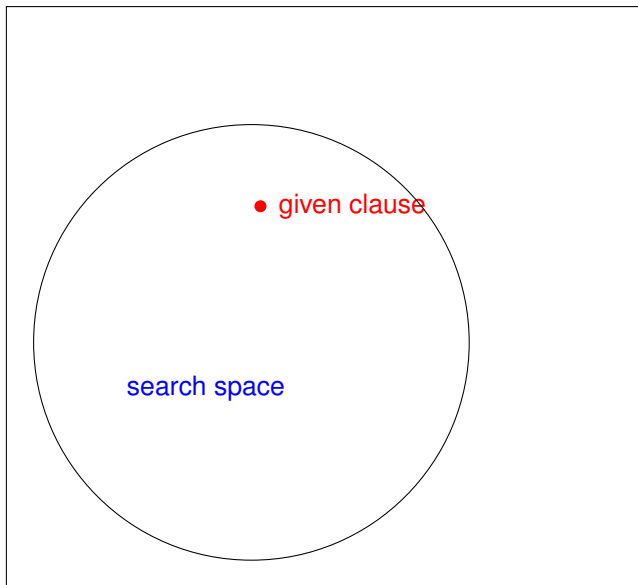
# Fair Saturation Algorithms: Inference Selection by Clause Selection



# Fair Saturation Algorithms: Inference Selection by Clause Selection

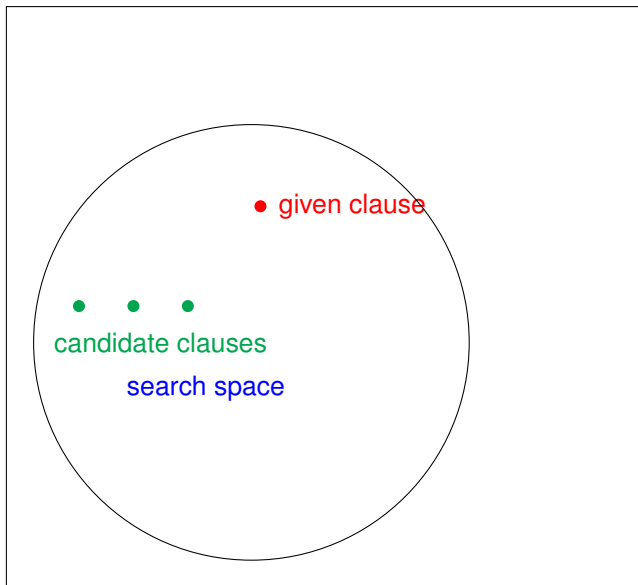


# Fair Saturation Algorithms: Inference Selection by Clause Selection

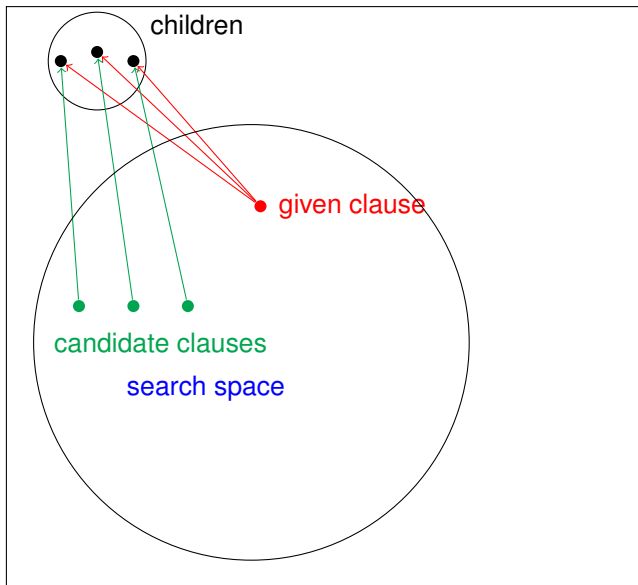




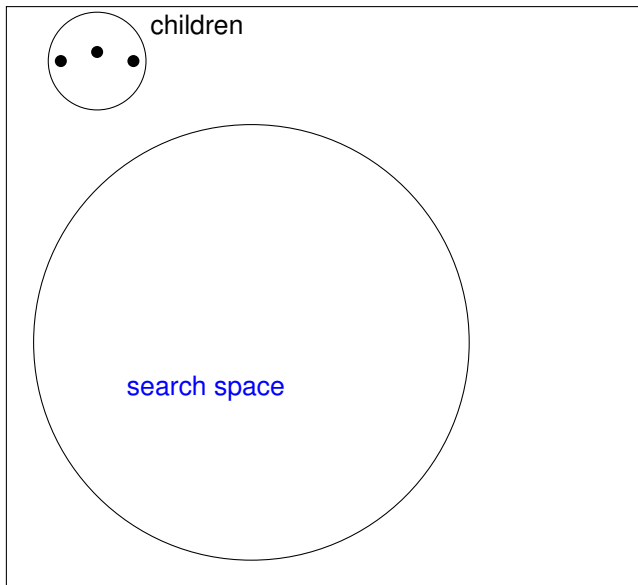
# Fair Saturation Algorithms: Inference Selection by Clause Selection



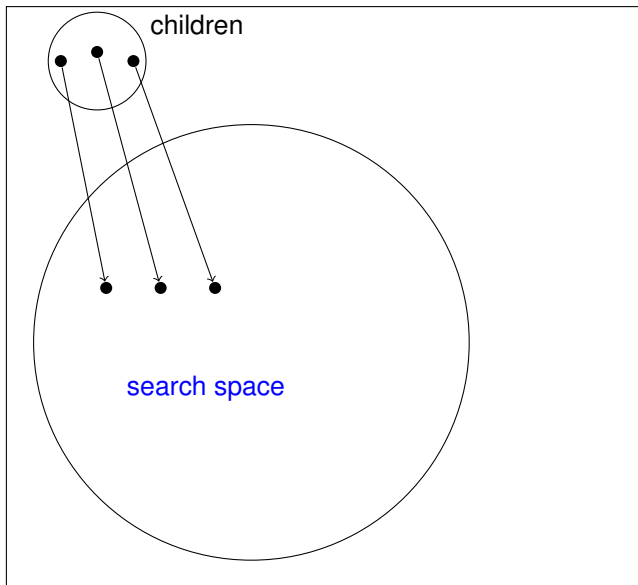
# Fair Saturation Algorithms: Inference Selection by Clause Selection



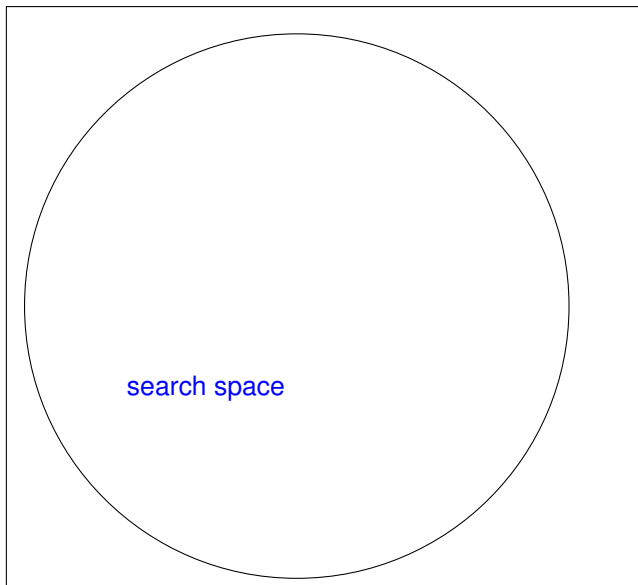
# Fair Saturation Algorithms: Inference Selection by Clause Selection



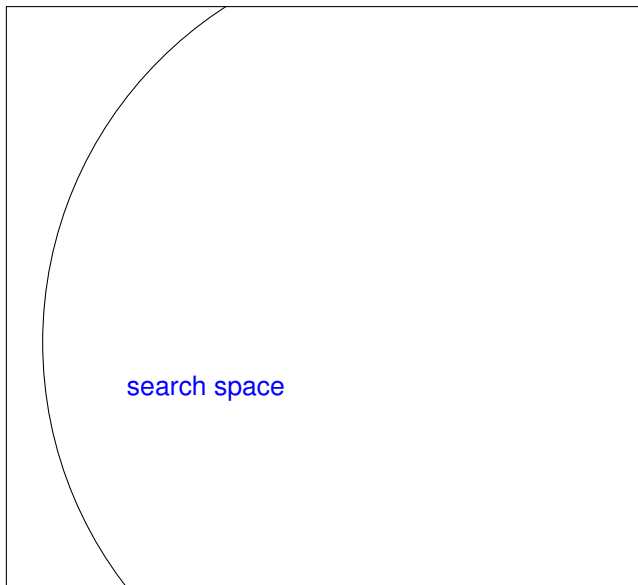
# Fair Saturation Algorithms: Inference Selection by Clause Selection



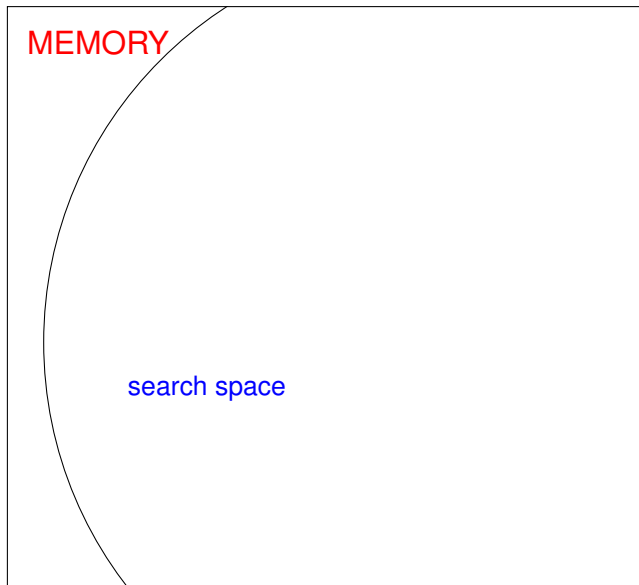
# Fair Saturation Algorithms: Inference Selection by Clause Selection



# Fair Saturation Algorithms: Inference Selection by Clause Selection



# Fair Saturation Algorithms: Inference Selection by Clause Selection



# Saturation Algorithm

A **saturation algorithm** tries to **saturate** a set of clauses with respect to a given inference system.

**In theory** there are three possible scenarios:

1. At some moment the empty clause  $\square$  is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating  $\square$ , in this case the input set of clauses is satisfiable.
3. Saturation will run **forever**, but without generating  $\square$ . In this case the input set of clauses is satisfiable.



# Saturation Algorithm in Practice

In practice there are three possible scenarios:

1. At some moment the empty clause  $\square$  is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating  $\square$ , in this case the input set of clauses is satisfiable.
3. Saturation will run until we run out of resources, but without generating  $\square$ . In this case it is unknown whether the input set is unsatisfiable.

# Saturation Algorithm

Even when we implement inference selection by clause selection, there are **too many inferences**, especially when the search space grows.

# Saturation Algorithm

Even when we implement inference selection by clause selection, there are **too many inferences**, especially when the search space grows.

**Solution:** only apply inferences to the **selected clause and the previously selected clauses**.

# Saturation Algorithm

Even when we implement inference selection by clause selection, there are **too many inferences**, especially when the search space grows.

**Solution:** only apply inferences to the **selected clause and the previously selected clauses**.

Thus, the search space is divided in two parts:

- ▶ **active clauses**, that participate in inferences;
- ▶ **passive clauses**, that do not participate in inferences.

# Saturation Algorithm

Even when we implement inference selection by clause selection, there are **too many inferences**, especially when the search space grows.

**Solution:** only apply inferences to the **selected clause and the previously selected clauses**.

Thus, the search space is divided in two parts:

- ▶ **active clauses**, that participate in inferences;
- ▶ **passive clauses**, that do not participate in inferences.

**Observation:** the set of passive clauses is usually considerably larger than the set of active clauses, often by 2-4 orders of magnitude (depending on the saturation algorithm and the problem).

# Saturation Algorithm in Practice

unprocessed clauses and kept (active and passive) clauses

```
--saturation_algorithm {lrs, otter, discount}
```

# Outline

Introduction

First-Order Logic and TPTP

Inference Systems

Saturation Algorithms

**Redundancy Elimination**

Equality

# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form  $A \vee \neg A \vee C$ , that is, it contains a pair of complementary literals. There are also **equational tautologies**, for example  $a \neq b \vee b \neq c \vee f(c, c) \simeq f(a, a)$ .



# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form  $A \vee \neg A \vee C$ , that is, it contains a pair of complementary literals.

There are also **equational tautologies**, for example  $a \neq b \vee b \neq c \vee f(c, c) \simeq f(a, a)$ .

A clause  $C$  **subsumes** any clause  $C \vee D$ , where  $D$  is non-empty.

# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form  $A \vee \neg A \vee C$ , that is, it contains a pair of complementary literals.

There are also **equational tautologies**, for example  $a \neq b \vee b \neq c \vee f(c, c) \simeq f(a, a)$ .

A clause  $C$  **subsumes** any clause  $C \vee D$ , where  $D$  is non-empty.

It was known since 1965 that **subsumed clauses and propositional tautologies can be removed from the search space.**

# Problem

How can we **prove** that **completeness is preserved** if we **remove subsumed clauses and tautologies** from the **search space**?

# Problem

How can we **prove** that **completeness is preserved** if we **remove subsumed clauses and tautologies** from the **search space**?

Solution: general **theory of redundancy**.

# Clause Orderings

The order  $\succ$  on atoms has already been extended to **literals**.

It can also be extended to a well-founded ordering  $\succ$  on **clauses** (using the finite multiset extension of the literal ordering).

This gives us a **reduction ordering to compares clauses**.

# Redundancy

A clause  $C \in S$  is called **redundant in  $S$**  if it is a logical consequence of clauses in  $S$  strictly smaller than  $C$ .

# Examples

A **tautology**  $A \vee \neg A \vee C$  is a logical consequence of the empty set of formulas:

$$\models A \vee \neg A \vee C,$$

therefore it is **redundant**.

# Examples

A **tautology**  $A \vee \neg A \vee C$  is a logical consequence of the empty set of formulas:

$$\models A \vee \neg A \vee C,$$

therefore it is **redundant**.

We know that  $C$  **subsumes**  $C \vee D$ . Note

$$\begin{aligned} C \vee D &\succ C \\ C &\models C \vee D \end{aligned}$$

therefore subsumed clauses are **redundant**.



# Examples

A **tautology**  $A \vee \neg A \vee C$  is a logical consequence of the empty set of formulas:

$$\models A \vee \neg A \vee C,$$

therefore it is **redundant**.

We know that  $C$  **subsumes**  $C \vee D$ . Note

$$\begin{aligned} C \vee D &\succ C \\ C &\models C \vee D \end{aligned}$$

therefore subsumed clauses are **redundant**.

If  $\square \in S$ , then all non-empty other clauses in  $S$  are **redundant**.

# Redundant Clauses Can be Removed

In  $\mathbb{BR}_\sigma$  (and in all calculi we will consider later) **redundant clauses can be removed from the search space.**

# Redundant Clauses Can be Removed

In  $\text{BR}_\sigma$  (and in all calculi we will consider later) **redundant clauses can be removed from the search space.**

# Inference Process with Redundancy

Let  $\mathbb{I}$  be an inference system. Consider an inference process with two kinds of step  $S_i \Rightarrow S_{i+1}$ :

1. Adding the conclusion of an  $\mathbb{I}$ -inference with premises in  $S_i$ .
2. Deletion of a clause redundant in  $S_i$ , that is

$$S_{i+1} = S_i - \{C\},$$

where  $C$  is redundant in  $S_i$ .

# Fairness: Persistent Clauses and Limit

Consider an inference process

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$$

A clause  $C$  is called **persistent** if

$$\exists \forall j \geq i (C \in S_j).$$

The **limit**  $S_\omega$  of the inference process is the set of all persistent clauses:

$$S_\omega = \bigcup_{i=0,1,\dots} \bigcap_{j \geq i} S_j.$$

# Fairness

The process is called  $\mathbb{I}$ -fair if every inference with persistent premises in  $S_\omega$  has been applied, that is, if

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

is an inference in  $\mathbb{I}$  and  $\{C_1, \dots, C_n\} \subseteq S_\omega$ , then  $C \in S_i$  for some  $i$ .

# Completeness of $\text{BR}_{\succ, \sigma}$

**Completeness Theorem.** Let  $\succ$  be a simplification ordering and  $\sigma$  a well-behaved selection function. Let also

1.  $S_0$  be a set of clauses;
2.  $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots$  be a fair  $\text{BR}_{\succ, \sigma}$ -inference process.

Then  $S_0$  is unsatisfiable if and only if  $\square \in S_i$  for some  $i$ .

# Saturation up to Redundancy

A set  $S$  of clauses is called **saturated up to redundancy** if for every  $\mathbb{I}$ -inference

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

with premises in  $S$ , either

1.  $C \in S$ ; or
2.  $C$  is redundant w.r.t.  $S$ , that is,  $S_{\setminus C} \models C$ .



# Saturation up to Redundancy and Satisfiability Checking

**Lemma.** A set  $S$  of clauses saturated up to redundancy in  $\text{BR}_{\gamma, \sigma}$  is unsatisfiable if and only if  $\square \in S$ .

# Saturation up to Redundancy and Satisfiability Checking

**Lemma.** A set  $S$  of clauses saturated up to redundancy in  $\mathbb{BR}_{\gamma, \sigma}$  is unsatisfiable if and only if  $\square \in S$ .

Therefore, if we built a set saturated up to redundancy, then the initial set  $S_0$  is **satisfiable**. This is a powerful way of checking redundancy: one can even check satisfiability of formulas having only **infinite models**.

# Binary Resolution with Selection

One of the **key properties** to satisfy this lemma is the following: the conclusion of every rule is strictly smaller than the rightmost premise of this rule.

- ▶ Binary resolution,

$$\frac{\underline{p} \vee C_1 \quad \underline{\neg p} \vee C_2}{C_1 \vee C_2} \text{ (BR).}$$

- ▶ Positive factoring,

$$\frac{\underline{p} \vee \underline{p} \vee C}{p \vee C} \text{ (Fact).}$$

# Outline

Introduction

First-Order Logic and TPTP

Inference Systems

Saturation Algorithms

Redundancy Elimination

**Equality**

# First-order logic with equality

- ▶ Equality predicate:  $=$ .
- ▶ Equality:  $l = r$ .

The order of literals in equalities does not matter, that is, we consider an equality  $l = r$  as a multiset consisting of two terms  $l, r$ , and so consider  $l = r$  and  $r = l$  equal.

# Equality. An Axiomatisation

- ▶ **reflexivity** axiom:  $x = x$ ;
- ▶ **symmetry** axiom:  $x = y \rightarrow y = x$ ;
- ▶ **transitivity** axiom:  $x = y \wedge y = z \rightarrow x = z$ ;
- ▶ **function substitution** axioms:  
 $x_1 = y_1 \wedge \dots \wedge x_n = y_n \rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$ , for every function symbol  $f$ ;
- ▶ **predicate substitution** axioms:  
 $x_1 = y_1 \wedge \dots \wedge x_n = y_n \wedge P(x_1, \dots, x_n) \rightarrow P(y_1, \dots, y_n)$  for every predicate symbol  $P$ .

# Inference systems for logic with equality

We will define a **resolution and superposition inference system**. This system is **complete**. One can **eliminate redundancy** (but the literal ordering needs to satisfy additional properties).

# Inference systems for logic with equality

We will define a **resolution and superposition inference system**. This system is **complete**. One can **eliminate redundancy** (but the literal ordering needs to satisfy additional properties).

Moreover, we will first define it only for ground clauses. On the theoretical side,

- ▶ Completeness is first proved for **ground clauses** only.
- ▶ It is then “lifted” to arbitrary clauses using a technique called **lifting**.
- ▶ Moreover, this way some notions (ordering, selection function) can first be defined for ground clauses only and then it is relatively easy to see how to generalise them for non-ground clauses.



# Simple Ground Superposition Inference System

Superposition: (right and left)

$$\frac{l = r \vee C \quad s[l] = t \vee D}{s[r] = t \vee C \vee D} \text{ (Sup)}, \quad \frac{l = r \vee C \quad s[l] \neq t \vee D}{s[r] \neq t \vee C \vee D} \text{ (Sup)},$$

# Simple Ground Superposition Inference System

Superposition: (right and left)

$$\frac{l = r \vee C \quad s[l] = t \vee D}{s[r] = t \vee C \vee D} \text{ (Sup)}, \quad \frac{l = r \vee C \quad s[l] \neq t \vee D}{s[r] \neq t \vee C \vee D} \text{ (Sup)},$$

Equality Resolution:

$$\frac{s \neq s \vee C}{C} \text{ (ER)},$$

# Simple Ground Superposition Inference System

Superposition: (right and left)

$$\frac{l = r \vee C \quad s[l] = t \vee D}{s[r] = t \vee C \vee D} \text{ (Sup)}, \quad \frac{l = r \vee C \quad s[l] \neq t \vee D}{s[r] \neq t \vee C \vee D} \text{ (Sup)},$$

Equality Resolution:

$$\frac{s \neq s \vee C}{C} \text{ (ER)},$$

Equality Factoring:

$$\frac{s = t \vee s = t' \vee C}{s = t \vee t \neq t' \vee C} \text{ (EF)},$$

# Example

$$f(a) = a \vee g(a) = a$$

$$f(f(a)) = a \vee g(g(a)) \neq a$$

$$f(f(a)) \neq a$$

# Can this system be used for efficient theorem proving?

Not really. It has **too many inferences**. For example, from the clause  $f(a) = a$  we can derive any clause of the form

$$f^m(a) = f^n(a)$$

where  $m, n \geq 0$ .

# Can this system be used for efficient theorem proving?

Not really. It has **too many inferences**. For example, from the clause  $f(a) = a$  we can derive any clause of the form

$$f^m(a) = f^n(a)$$

where  $m, n \geq 0$ .

Worst of all, the derived clauses can be **much larger** than the original clause  $f(a) = a$ .

# Can this system be used for efficient theorem proving?

Not really. It has **too many inferences**. For example, from the clause  $f(a) = a$  we can derive any clause of the form

$$f^m(a) = f^n(a)$$

where  $m, n \geq 0$ .

Worst of all, the derived clauses can be **much larger** than the original clause  $f(a) = a$ .

The recipe is to use the previously introduced ingredients:

1. Ordering;
2. Literal selection;
3. Redundancy elimination.

# Ground Superposition Inference System $\text{Sup}_{\succ, \sigma}$

Let  $\sigma$  be a literal selection function.

Superposition: (right and left)

$$\frac{l = r \vee C \quad s[l] = t \vee D}{s[r] = t \vee C \vee D} \text{ (Sup)}, \quad \frac{l = r \vee C \quad s[l] \neq t \vee D}{s[r] \neq t \vee C \vee D} \text{ (Sup)},$$

where (i)  $l \succ r$ , (ii)  $s[l] \succ t$ , (iii)  $l = r$  is strictly greater than any literal in  $C$ , (iv)  $s[l] = t$  is greater than or equal to any literal in  $D$ .

Equality Resolution:

$$\frac{s \neq s \vee C}{C} \text{ (ER)},$$

Equality Factoring:

$$\frac{s = t \vee s = t' \vee C}{s = t \vee t \neq t' \vee C} \text{ (EF)},$$

where (i)  $s \succ t \succeq t'$ ; (ii)  $s = t$  is greater than or equal to any literal in  $C$ .



# Simplification Ordering

The only restriction we imposed on term orderings was **well-foundedness** and **stability under substitutions**. When we deal with equality, these two properties are insufficient. We need a third property, called **monotonicity**.

An ordering  $\succ$  on terms is called a **simplification ordering** if

1.  $\succ$  is **well-founded**;
2.  $\succ$  is **monotonic**: if  $l \succ r$ , then  $s[l] \succ s[r]$ ;
3.  $\succ$  is **stable under substitutions**: if  $l \succ r$ , then  $l\theta \succ r\theta$ .

# Simplification Ordering

The only restriction we imposed on term orderings was **well-foundedness** and **stability under substitutions**. When we deal with equality, these two properties are insufficient. We need a third property, called **monotonicity**.

An ordering  $\succ$  on terms is called a **simplification ordering** if

1.  $\succ$  is **well-founded**;
2.  $\succ$  is **monotonic**: if  $l \succ r$ , then  $s[l] \succ s[r]$ ;
3.  $\succ$  is **stable under substitutions**: if  $l \succ r$ , then  $l\theta \succ r\theta$ .

One can combine the last two properties into one:

- 2a. If  $l \succ r$ , then  $s[l\theta] \succ s[r\theta]$ .

# A General Property of Term Orderings

If  $\succ$  is a simplification ordering, then for every term  $t[s]$  and its proper subterm  $s$  we have  $s \not\succeq t[s]$ .

# A General Property of Term Orderings

If  $\succ$  is a simplification ordering, then for every term  $t[s]$  and its proper subterm  $s$  we have  $s \not\succeq t[s]$ .

Consider an example.

$$f(a) = a$$

$$f(f(a)) = a$$

$$f(f(f(a))) = a$$

Then both  $f(f(a)) = a$  and  $f(f(f(a))) = a$  are **redundant**. The clause  $f(a) = a$  is a logical consequence of  $\{f(f(a)) = a, f(f(f(a))) = a\}$  but is **not redundant**.

## New redundancy

Consider a superposition with a unit left premise:

$$\frac{l = r \quad s[l] = t \vee D}{s[r] = t \vee D} \text{ (Sup),}$$

Note that we have

$$l = r, s[r] = t \vee D \vDash s[l] = t \vee D$$

## New redundancy

Consider a superposition with a unit left premise:

$$\frac{l = r \quad s[l] = t \vee D}{s[r] = t \vee D} \text{ (Sup),}$$

Note that we have

$$l = r, s[r] = t \vee D \vDash s[l] = t \vee D$$

and we have

$$s[l] = t \vee D \succ s[r] = t \vee D.$$

## New redundancy

Consider a superposition with a unit left premise:

$$\frac{l = r \quad s[l] = t \vee D}{s[r] = t \vee D} \text{ (Sup),}$$

Note that we have

$$l = r, s[r] = t \vee D \vDash s[l] = t \vee D$$

and we have

$$s[l] = t \vee D \succ s[r] = t \vee D.$$

If we also have  $l = r \succ s[r] = t \vee D$ , then the second premise is **redundant** and can be removed.

## New redundancy

Consider a superposition with a unit left premise:

$$\frac{l = r \quad s[l] = t \vee D}{s[r] = t \vee D} \text{ (Sup),}$$

Note that we have

$$l = r, s[r] = t \vee D \vDash s[l] = t \vee D$$

and we have

$$s[l] = t \vee D \succ s[r] = t \vee D.$$

If we also have  $l = r \succ s[r] = t \vee D$ , then the second premise is **redundant** and can be removed.

This rule (superposition plus deletion) is sometimes called **demodulation** (also **rewriting by unit equalities**).