First-Order Theorem Proving

Laura Kovács and Andrei Voronkov
TU Wien and U. Manchester and EasyChair

Outline

First-Order Theorem Proving

We will use the Vampire theorem prover throughout the lecture.

Go to

`https://vprover.github.io/download.html`

and pick the route most suitable to you.

Notes:

- For Linux users, a binary is probably the easiest route
- For Mac users, you need to build from source
  - run `make vampire_rel`
- For Windows users, the easiest route is to thank Geoff and use

  `https://www.tptp.org/cgi-bin/SystemOnTPTP`

First-Order Theorem Proving

We will use the VAMPIRE theorem prover throughout the lecture.

Go to

`https://vprover.github.io/download.html`

and pick the route most suitable to you.

Notes:

- For Linux users, a binary is probably the easiest route
- For Mac users, you need to build from source
  - run `make vampire_rel`
- For Windows users, the easiest route is to thank Geoff and use

  `https://www.tptp.org/cgi-bin/SystemOnTPTP`

Outline

First-Order Theorem Proving. An Example

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

First-Order Theorem Proving. An 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. An 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.

$$\begin{array}{l} \forall x(1 \cdot x = x) \\ \forall x(x^{-1} \cdot x = 1) \\ \forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z)) \end{array}$$

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 (Thousands of Problems for Theorem Provers), `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 (Thousands of Problems for Theorem Provers), `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 = 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] : mult(X,Y) = mult(Y,X)).
```

Running Vampire on a TPTP file

is easy: simply use

```
vampire <filename>
```

Running Vampire on 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 TUWien group.tptp
```

Proof by Vampire (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 (Slightliy Modified)

```
Refutation found.
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 . . .

Outline

Setting the Scene

First-Order Theorem Proving - An Example

First-Order Logic and TPTP

Inference Systems

Selection Functions

Saturation Algorithms

Redundancy Elimination

Equality

Term Orderings

Completeness of Ground Superposition

Unification and Lifting

Non-Ground Superposition

First-Order Logic and TPTP – Recap

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

First-Order Logic and TPTP – Recap

- 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 – Recap

- 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 $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms.

First-Order Logic and TPTP – Recap

- 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 $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms. Terms denote domain elements.

First-Order Logic and TPTP – Recap

- 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 – Recap

- 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 $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms. Terms denote domain elements.
- Atomic formula: expression $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms. Formulas denote properties of domain elements.
- All symbols are uninterpreted, apart from equality $=$.

First-Order Logic and TPTP – Recap

- 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 $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms. Terms denote domain elements.
- Atomic formula: expression $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms. Formulas denote properties of domain elements.
- All symbols are uninterpreted, apart from equality $=$.

FOL	TPTP
$\bot, \top$	`$false, $true`
$\neg a$	`~a`
$a_1 \wedge \ldots \wedge a_n$	`a1 & ... & an`
$a_1 \vee \ldots \vee a_n$	`a1 | ... | an`
$a_1 \rightarrow a_2$	`a1 => a2`
$(\forall x_1) \ldots (\forall x_n) a$	`! [X1,...,Xn] : a`
$(\exists x_1) \ldots (\exists x_n) a$	`? [X1,...,Xn] : a`

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.
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.
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.
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.
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.
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.
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.
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.
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.
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 $>$ 50 times.

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 $F_1, \ldots, F_n$ and conjecture G,

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

Proof by Refutation

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

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

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

Proof by Refutation

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

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

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 *false*.
- If *false* 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 *false*.
- If *false* is derived, report the result, maybe including a refutation.

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

Outline

- Setting the Scene
- First-Order Theorem Proving - An Example
- First-Order Logic and TPTP
- Inference Systems
- Selection Functions
- Saturation Algorithms
- Redundancy Elimination
- Equality
- Term Orderings
- Completeness of Ground Superposition
- Unification and Lifting
- Non-Ground Superposition

Inference System

- inference has the form

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

  where $n \geq 0$ and $F_1, \ldots, F_n, G$ are formulas.
- The formula G is called the conclusion of the inference;
- The formulas $F_1, \ldots, F_n$ are called its premises.
- An inference rule R is a set of inferences.
- Every inference $I \in R$ is called an instance of R.
- An Inference system $\mathbb{I}$ is a set of inference rules.
- Axiom: inference rule with no premises.

Inference System: Example

Represent the natural number n by the string $\underbrace{|\ldots|}_{n \text{ times}}\varepsilon$.

The following inference system contains 6 inference rules for deriving equalities between expressions containing natural numbers, addition $+$ and multiplication $\cdot$.

$$\frac{}{\varepsilon = \varepsilon}\ (\varepsilon) \qquad\qquad \frac{x = y}{|x = |y}\ (|)$$

$$\frac{}{\varepsilon + x = x}\ (+_1) \qquad\qquad \frac{x + y = z}{|x + y = |z}\ (+_2)$$

$$\frac{}{\varepsilon \cdot x = \varepsilon}\ (\cdot_1) \qquad\qquad \frac{x \cdot y = u \quad y + u = z}{|x \cdot y = z}\ (\cdot_2)$$

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 $E_1, \ldots, E_m$: a finite derivation of E whose every leaf is either an axiom or one of the expressions $E_1, \ldots, E_m$.

Examples

For example,

$$\frac{||\varepsilon + |\varepsilon = |||\varepsilon}{|||\varepsilon + |\varepsilon = ||||\varepsilon}\ (+_2)$$

is an inference that is an instance (special case) of the inference rule

$$\frac{x + y = z}{|x + y = |z}\ (+_2)$$

Examples

For example,

$$\frac{||\varepsilon + |\varepsilon = |||\varepsilon}{|||\varepsilon + |\varepsilon = ||||\varepsilon}\ (+_2)$$

is an inference that is an instance (special case) of the inference rule

$$\frac{x + y = z}{|x + y = |z}\ (+_2)$$

It has one premise $||\varepsilon + |\varepsilon = |||\varepsilon$ and the conclusion $|||\varepsilon + |\varepsilon = ||||\varepsilon$.

Examples

For example,

$$\frac{||\varepsilon + |\varepsilon = |||\varepsilon}{|||\varepsilon + |\varepsilon = ||||\varepsilon}\ (+_2)$$

is an inference that is an instance (special case) of the inference rule

$$\frac{x + y = z}{|x + y = |z}\ (+_2)$$

It has one premise $||\varepsilon + |\varepsilon = |||\varepsilon$ and the conclusion $|||\varepsilon + |\varepsilon = ||||\varepsilon$.

The axiom

$$\frac{}{\varepsilon + |||\varepsilon = |||\varepsilon}\ (+_1)$$

is an instance of the rule

$$\frac{}{\varepsilon + x = x}\ (+_1)$$

Proof in this Inference System

Proof of $||\varepsilon \cdot ||\varepsilon = ||||\varepsilon$ (that is, $2 \cdot 2 = 4$).

$$\dfrac{\dfrac{\dfrac{}{\varepsilon \cdot ||\varepsilon = \varepsilon}\,(\cdot_1) \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon + \varepsilon = \varepsilon}\,(+_1)}{|\varepsilon + \varepsilon = |\varepsilon}\,(+_2)}{||\varepsilon + \varepsilon = ||\varepsilon}\,(+_2)}{|\varepsilon \cdot ||\varepsilon = ||\varepsilon}\,(\cdot_2) \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon + ||\varepsilon = ||\varepsilon}\,(+_1)}{|\varepsilon + ||\varepsilon = |||\varepsilon}\,(+_2)}{||\varepsilon + ||\varepsilon = ||||\varepsilon}\,(+_2)}{||\varepsilon \cdot ||\varepsilon = ||||\varepsilon}\,(\cdot_2).$$

Proof, Derivation in this Inference System

Proof of $||\varepsilon \cdot ||\varepsilon = ||||\varepsilon$ (that is, $2 \cdot 2 = 4$).

Derivation of $|\varepsilon \cdot ||\varepsilon = ||\varepsilon$ from $\varepsilon \cdot ||\varepsilon = \varepsilon$ and $|\varepsilon + \varepsilon = |\varepsilon$.

$$\dfrac{\dfrac{\dfrac{}{\varepsilon \cdot ||\varepsilon = \varepsilon}\,(\cdot_1) \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon + \varepsilon = \varepsilon}\,(+_1)}{|\varepsilon + \varepsilon = |\varepsilon}\,(+_2)}{||\varepsilon + \varepsilon = ||\varepsilon}\,(+_2)}{|\varepsilon \cdot ||\varepsilon = ||\varepsilon}\,(\cdot_2) \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon + ||\varepsilon = ||\varepsilon}\,(+_1)}{|\varepsilon + ||\varepsilon = |||\varepsilon}\,(+_2)}{||\varepsilon + ||\varepsilon = ||||\varepsilon}\,(+_2)}{||\varepsilon \cdot ||\varepsilon = ||||\varepsilon}\,(\cdot_2).$$

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 buit 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 $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.

Clauses

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

Clauses

- Literal: either an atom A or its negation $\neg A$.
- Clause: a disjunction $L_1 \lor \ldots \lor L_n$ 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 $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\Box$: clause with 0 literals, that is, when $n = 0$.
- A formula in Clausal Normal Form (CNF): a conjunction of clauses.
- From now on: 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 $\mathbb{BR}$ is an inference system on propositional clauses (or ground clauses). It consists of two inference rules:

- Binary resolution, denoted by BR:

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

- Factoring, denoted by Fact:

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

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:

$$\dfrac{\dfrac{\dfrac{p \vee q \qquad p \vee \neg q}{p \vee p}\ (\mathrm{BR})}{p}\ (\mathrm{Fact}) \qquad \dfrac{\dfrac{\neg p \vee q \qquad \neg p \vee \neg q}{\neg p \vee \neg p}\ (\mathrm{BR})}{\neg p}\ (\mathrm{Fact})}{\Box}\ (\mathrm{BR})$$

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 $\mathbb{BR}$.

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 $\mathbb{BR}$.
2. We have to formalize search for derivations.

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

Outline

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.,

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

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.,

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

Note: selection function does not have to be a function. It can be any oracle that selects literals.

Binary Resolution with Selection

We introduce a family of inference systems, parametrised by a literal selection function σ.

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

- Binary resolution, denoted by BR

$$\frac{\underline{p} \vee C_1 \qquad \underline{\neg p} \vee C_2}{C_1 \vee C_2}\ (\mathsf{BR}).$$

Binary Resolution with Selection

We introduce a family of inference systems, parametrised by a literal selection function σ.
The binary resolution inference system, denoted by $\mathbb{BR}_\sigma$, consists of two inference rules:

- Binary resolution, denoted by BR

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

- Positive factoring, denoted by Fact:

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

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:

$$
\begin{array}{ll}
(1) & \neg q \lor \underline{r} \\
(2) & \neg p \lor \underline{q} \\
(3) & \neg r \lor \underline{\neg q} \\
(4) & \neg q \lor \underline{\neg p} \\
(5) & \neg p \lor \underline{\neg r} \\
(6) & \neg r \lor \underline{p} \\
(7) & r \lor q \lor \underline{p}
\end{array}
$$

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:

$$A_0 \succ A_1 \succ A_2 \succ \cdots$$

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:

$$A_0 \succ A_1 \succ A_2 \succ \cdots$$

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 $\neg p \succ q$;
- $\neg p \succ p$.

Literal Orderings

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

$$A_0 \succ A_1 \succ A_2 \succ \cdots$$

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 $\neg p \succ q$;
- $\neg p \succ p$.

Exercise: prove that the induced ordering on literals is well-founded too.

Orderings and Well-Behaved Selections

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

- either a negative literal is selected,
  or all maximal literals (w.r.t. $\succ$) must be selected in C.

Orderings and Well-Behaved Selections

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

- either a negative literal is selected,
  or all maximal literals (w.r.t. $\succ$) must be selected in C.

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

A well-behave 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.

Checking (un)satisfiability – Where we are:

1. Completeness.
   Let S be an unsatisfiable set of clauses. Then there exists a derivation of $\Box$ from S in $\mathbb{BR}$.

.

Checking (un)satisfiability – Where we are:

1. Completeness.
   Let S be an unsatisfiable set of clauses. Then there exists a derivation of $\Box$ from S in $\mathbb{BR}$.
2. We have to formalize search for derivations.

.

Checking (un)satisfiability – Where we are:

1. Completeness.
   Let S be an unsatisfiable set of clauses. Then there exists a derivation of $\Box$ from S in $\mathbb{BR}$.
2. We have to formalize search for derivations.

We introduced well-behaved selection functions for selecting literals in clauses and applying inferences only over selected literals.

Binary resolution $\mathbb{BR}$ with selection is complete for every well-behaved selection function.

End of Lecture 1

Slides for lecture 1 ended here . . .

Outline

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 formulations 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 formulations 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 $S_0, S_1, \ldots$, denoted by

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$$

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

Inference Process

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

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$$

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

We say that this step is an $\mathbb{I}$-step if

1. there exists an inference
$$\frac{F_1 \quad \ldots \quad F_n}{F}$$
in $\mathbb{I}$ such that $\{F_1, \ldots, F_n\} \subseteq S_i$;
2. $S_{i+1} = S_i \cup \{F\}$.

Inference Process

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

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$$

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

We say that this step is an $\mathbb{I}$-step if

1. there exists an inference
$$\frac{F_1 \quad \ldots \quad F_n}{F}$$
in $\mathbb{I}$ such that $\{F_1, \ldots, F_n\} \subseteq S_i$;
2. $S_{i+1} = S_i \cup \{F\}$.

An $\mathbb{I}$-inference process is an inference process whose every step is an $\mathbb{I}$-step.

Property

Let $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ 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 $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ is the set of formulas $\bigcup_i S_i$.

Limit of a Process

The limit of an inference process $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ 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 $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ 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 the binary resolution inference system.

Limit of a Process

The limit of an inference process $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ 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 the binary resolution inference system.

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

Fairness

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

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

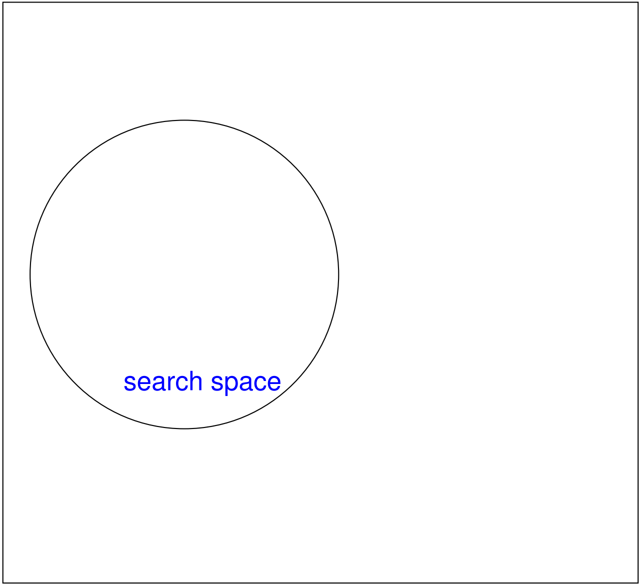
if $\{F_1, \ldots, F_n\} \subseteq S_\omega$, then there exists i such that $F \in S_i$.

Completeness, reformulated

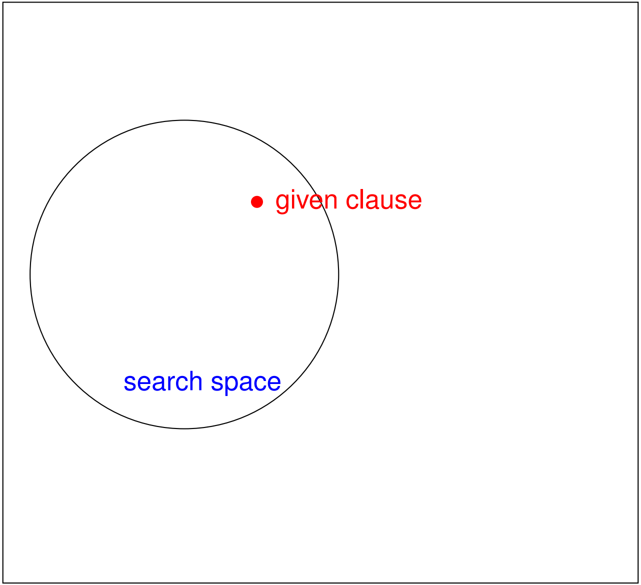
Theorem Let $\mathbb{I}$ be an inference system. The following conditions are equivalent.

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 $\square$.

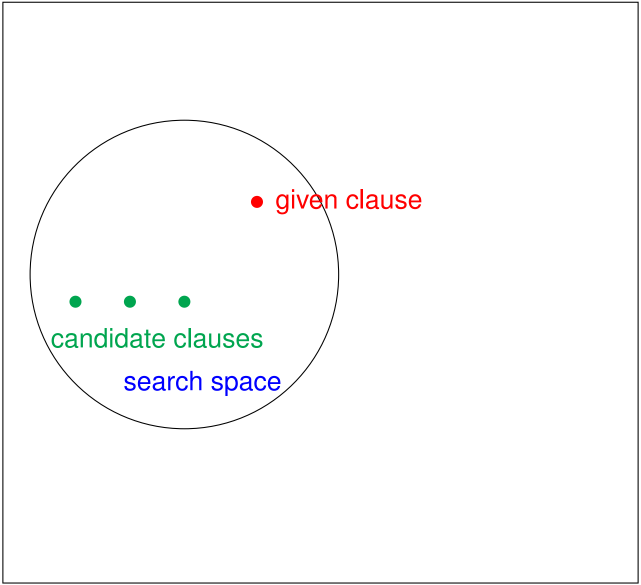
Fair Saturation Algorithms: Inference Selection by Clause Selection



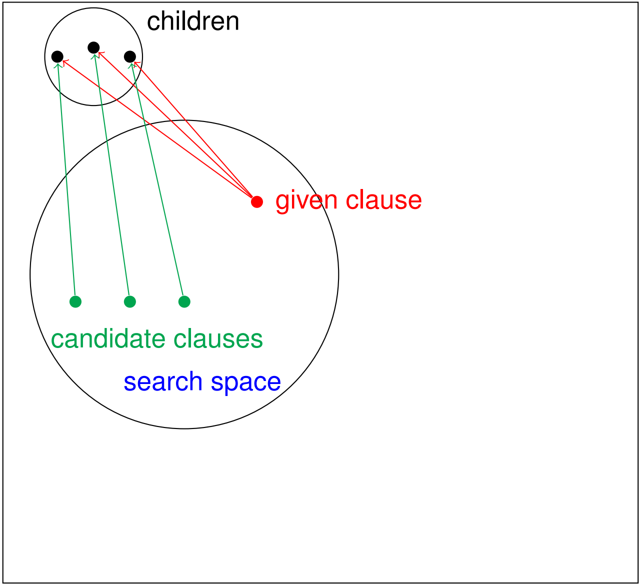
Fair Saturation Algorithms: Inference Selection by Clause Selection



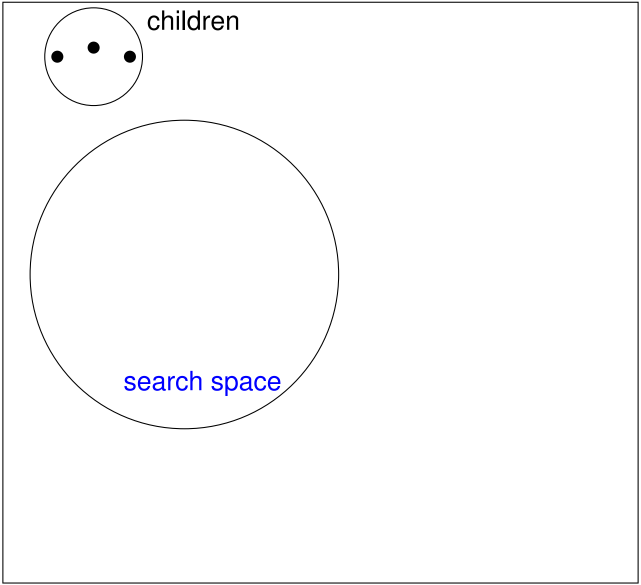
Fair Saturation Algorithms: Inference Selection by Clause Selection



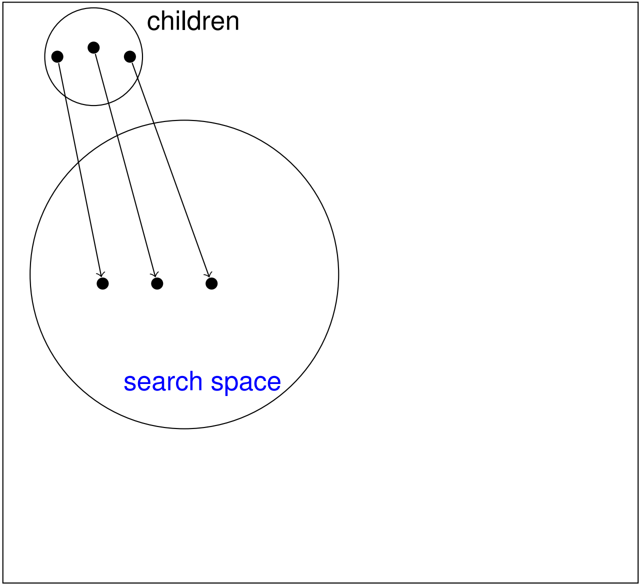
Fair Saturation Algorithms: Inference Selection by Clause Selection



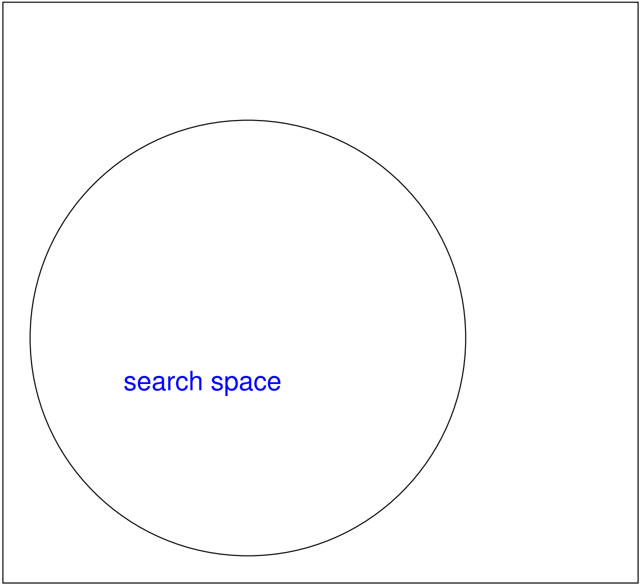
Fair Saturation Algorithms: Inference Selection by Clause Selection



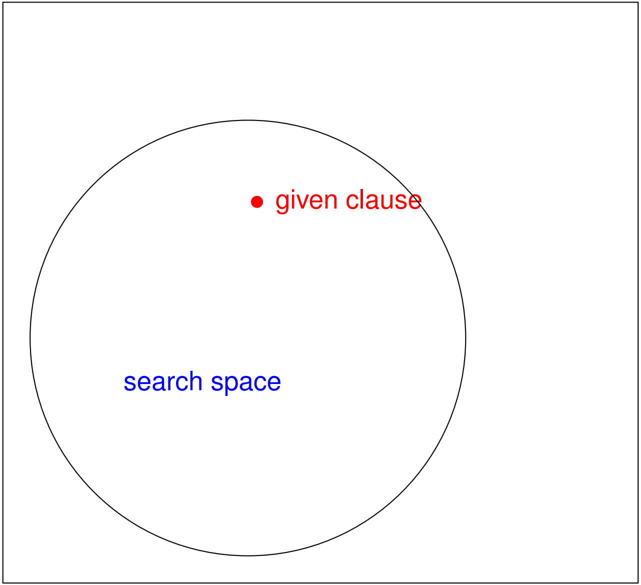
Fair Saturation Algorithms: Inference Selection by Clause Selection



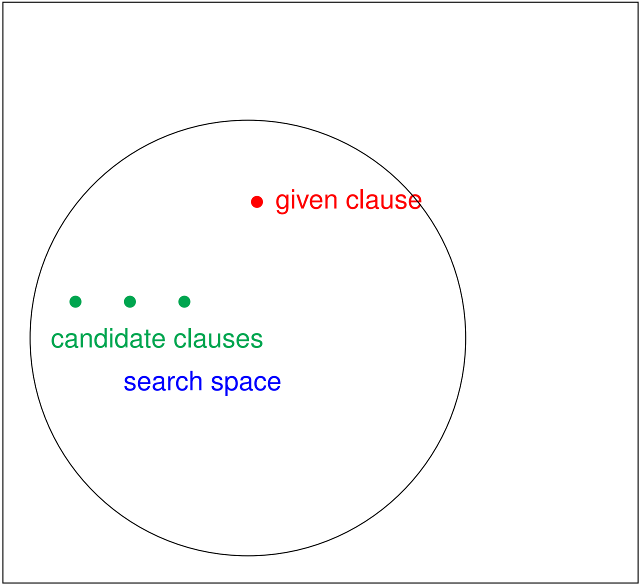
Fair Saturation Algorithms: Inference Selection by Clause Selection



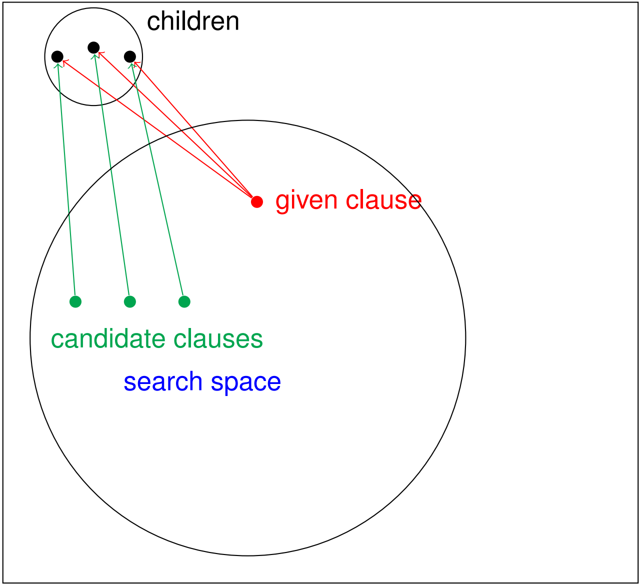
Fair Saturation Algorithms: Inference Selection by Clause Selection



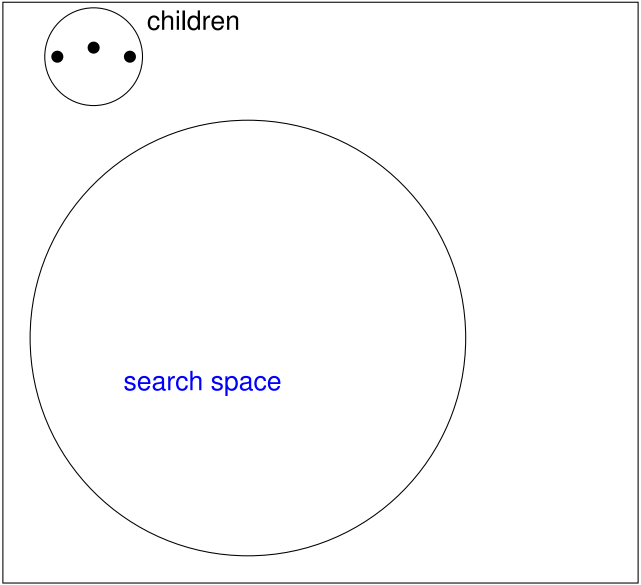
Fair Saturation Algorithms: Inference Selection by Clause Selection



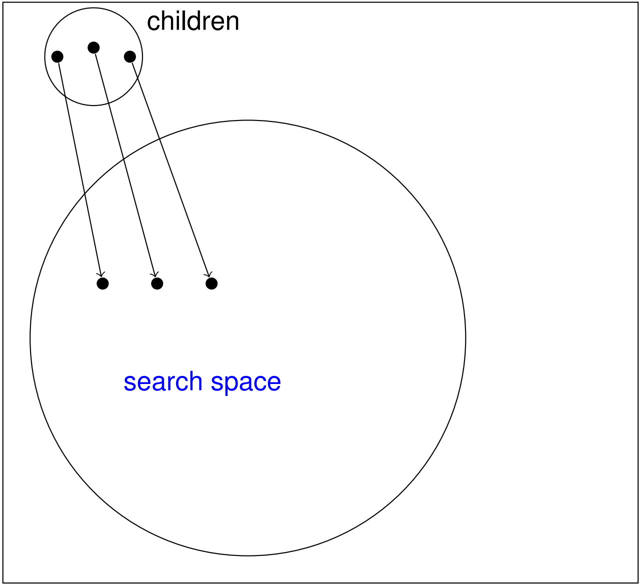
Fair Saturation Algorithms: Inference Selection by Clause Selection



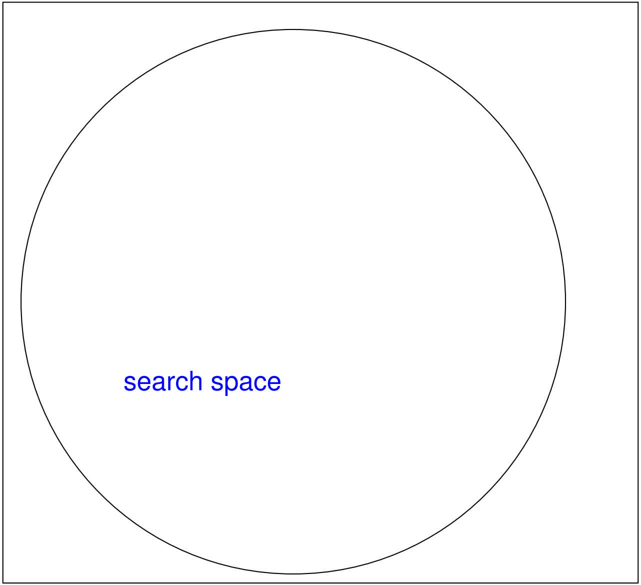
Fair Saturation Algorithms: Inference Selection by Clause Selection



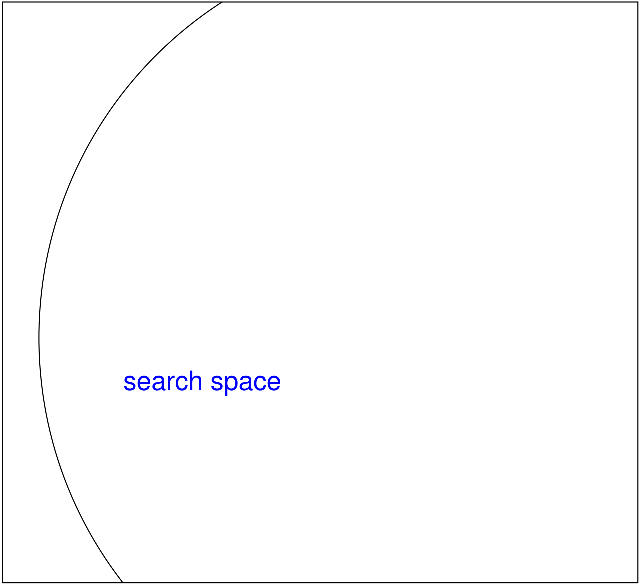
Fair Saturation Algorithms: Inference Selection by Clause Selection



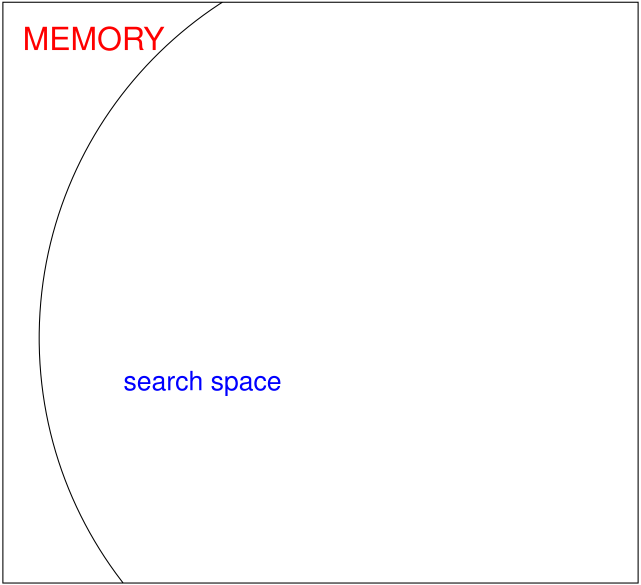
Fair Saturation Algorithms: Inference Selection by Clause Selection



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 in 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 in 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.

Outline

Subsumption and Tautology Deletion

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

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

Subsumption and Tautology Deletion

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

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

Subsumption and Tautology Deletion

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

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.

Bag Extension of an Ordering

Bag = finite multiset.
Let $>$ be any (strict) ordering on a set X. The bag extension of $>$ is a binary relation $>^{bag}$, on bags over X, defined as the smallest transitive relation on bags such that

$$\{x, y_1, \ldots, y_n\} >^{bag} \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$$
$$\text{if } x > x_i \text{ for all } i \in \{1 \ldots m\},$$

where $m \geq 0$.

Bag Extension of an Ordering

Bag = finite multiset.
Let $>$ be any (strict) ordering on a set X. The bag extension of $>$ is a binary relation $>^{bag}$, on bags over X, defined as the smallest transitive relation on bags such that

$$\{x, y_1, \ldots, y_n\} >^{bag} \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$$
$$\text{if } x > x_i \text{ for all } i \in \{1 \ldots m\},$$

where $m \geq 0$.
Idea: a bag becomes smaller if we replace an element by any finite number of smaller elements.

Bag Extension of an Ordering

Bag = finite multiset.
Let $>$ be any (strict) ordering on a set X. The bag extension of $>$ is a binary relation $>^{bag}$, on bags over X, defined as the smallest transitive relation on bags such that

$$\{x, y_1, \ldots, y_n\} >^{bag} \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$$
$$\text{if } x > x_i \text{ for all } i \in \{1 \ldots m\},$$

where $m \geq 0$.
Idea: a bag becomes smaller if we replace an element by any finite number of smaller elements.
The following results are known about the bag extensions of orderings:

1. $>^{bag}$ is an ordering;
2. If $>$ is total, then so is $>^{bag}$;
3. If $>$ is well-founded, then so is $>^{bag}$.

Clause Orderings

From now on consider clauses also as bags of literals. Note:

- we have an ordering $\succ$ for comparing literals;
- a clause is a bag of literals.

Clause Orderings

From now on consider clauses also as bags of literals. Note:

- we have an ordering $\succ$ for comparing literals;
- a clause is a bag of literals.

Hence

- we can compare clauses using the bag extension $\succ^{bag}$ of $\succ$.

Clause Orderings

From now on consider clauses also as bags of literals. Note:

- we have an ordering $\succ$ for comparing literals;
- a clause is a bag of literals.

Hence

- we can compare clauses using the bag extension $\succ^{bag}$ of $\succ$.

For simpicity we denote the multiset ordering also by $\succ$.

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 $p \lor \neg p \lor C$ is a logical consequence of the empty set of formulas:

$$\models p \lor \neg p \lor C,$$

therefore it is redundant.

Examples

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

$$\models p \vee \neg p \vee C,$$

therefore it is redundant.
We know that C subsumes $C \vee D$. Note

$$C \vee D \succ C$$
$$C \models C \vee D$$

therefore subsumed clauses are redundant.

Examples

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

$$\models p \vee \neg p \vee C,$$

therefore it is redundant.
We know that C subsumes $C \vee D$. Note

$$\begin{array}{c} C \vee D \succ C \\ C \models C \vee D \end{array}$$

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 $\mathbb{BR}_\sigma$ (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

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

The limit S_ω 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_ω has been applied, that is, if

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

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

Completeness of $\mathbb{BR}_\sigma$

Completeness Theorem. Let $\succ$ be a well-founded ordering and σ a well-behaved selection function. Let also

1. S_0 be a set of clauses;
2. $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ be a fair $\mathbb{BR}_\sigma$-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

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

with premises in S, either

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

Saturation up to Redundancy and Satisfiability Checking

Lemma. A set S of clauses saturated up to redundancy 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 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.

Saturation up to Redundancy and Satisfiability Checking

Lemma. A set S of clauses saturated up to redundancy 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.

The only problem with this characterisation is that there is no obvious way to build a model of S_0 out of a saturated set.

Binary Resolution with Selection

One of the key properties to satisfy this lemma is the following: the conclusion of every rule is strictly smaller that 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,

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

End of Lecture 2

Slides for lecture 2 ended here . . .

Outline

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 (Recap)

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

Inference systems for logic with equality

We will define a resolution and superposition inference system. This system is complete. One can eliminate redundancy.

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 first-order 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

$$\begin{array}{l} f(a) = a \vee g(a) = a \\ f(f(a)) = a \vee g(g(a)) \neq a \\ f(f(a)) \neq a \end{array}$$

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.

Atom and literal orderings on equalities

Equality atom comparison treats an equality $s = t$ as the multiset $\dot{\{} s, t \dot{\}}$.

- $(s' = t') \succ_{lit} (s = t)$ if $\dot{\{} s', t' \dot{\}} \succ \dot{\{} s, t \dot{\}}$
- $(s' \neq t') \succ_{lit} (s \neq t)$ if $\dot{\{} s', t' \dot{\}} \succ \dot{\{} s, t \dot{\}}$

with $\succ_{lit}$ being an induced ordering on literals.

Ground Superposition Inference System $\mathbb{S}\mathrm{up}_{\succ,\sigma}$

Let σ be a well-behaved literal selection function.
Superposition: (right and left)

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

where (i) $l \succ r$, (ii) $s[l] \succ t$

Ground Superposition Inference System $\mathbb{S}\mathrm{up}_{\succ,\sigma}$

Let σ be a well-behaved literal selection function.
Superposition: (right and left)

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

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

Ground Superposition Inference System $\mathbb{S}\mathrm{up}_{\succ,\sigma}$

Let σ be a well-behaved literal selection function.
Superposition: (right and left)

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

where (i) $l \succ r$, (ii) $s[l] \succ t$, (iii) $l = r$ is strictly greater than any literal in C, (iv) (only for the superposition-right rule) $s[l] = t$ is greater than or equal to any literal in D.
Equality Resolution:

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

Ground Superposition Inference System $\mathbb{S}\mathrm{up}_{\succ,\sigma}$

Let σ be a well-behaved literal selection function.

Superposition: (right and left)

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

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

Equality Resolution:

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

Equality Factoring:

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

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

Extension to arbitrary (non-equality) literals

- Consider a two-sorted logic in which equality is the only predicate symbol.
- Interpret terms as terms of the first sort and non-equality atoms as terms of the second sort.
- Add a constant $\top$ of the second sort.
- Replace non-equality atoms $p(t_1, \ldots, t_n)$ by equalities of the second sort $p(t_1, \ldots, t_n) = \top$.

Extension to arbitrary (non-equality) literals

- Consider a two-sorted logic in which equality is the only predicate symbol.
- Interpret terms as terms of the first sort and non-equality atoms as terms of the second sort.
- Add a constant $\top$ of the second sort.
- Replace non-equality atoms $p(t_1, \ldots, t_n)$ by equalities of the second sort $p(t_1, \ldots, t_n) = \top$.

For example, the clause

$$p(a, b) \vee \neg q(a) \vee a \neq b$$

becomes

$$p(a, b) = \top \vee q(a) \neq \top \vee a \neq b.$$

Binary resolution inferences can be represented by inferences in the superposition system

We ignore selection functions.

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

$$\frac{\dfrac{A = \top \vee C_1 \quad A \neq \top \vee C_2}{\top \neq \top \vee C_1 \vee C_2} \text{ (Sup)}}{C_1 \vee C_2} \text{ (ER)}$$

Exercise

Positive factoring can also be represented by inferences in the superposition system.

Outline

Simplification Ordering

When we deal with equality, we need to work with term orderings.
Consider a strict ordering $\succ$ on signature symbols, such that $\succ$ is well-founded.
The 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

When we deal with equality, we need to work with term orderings.
Consider a strict ordering $\succ$ on signature symbols, such that $\succ$ is well-founded.
The 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 $s \not\succ t[s]$. Why?

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 $s \not\succ t[s]$. Why?

Consider an example.

$$\begin{array}{l} f(a) = a \\ f(f(a)) = a \\ f(f(f(a))) = a \end{array}$$

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.

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 $s \not\succ t[s]$. Why?

Consider an example.

$$\begin{array}{l} f(a) = a \\ f(f(a)) = a \\ f(f(f(a))) = a \end{array}$$

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.

Exercise: Show that $\{f(a) = a, f(f(f(a))) \neq a\}$ is unsatisfiable, by using superposition with redundancy elimination.

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 $s \not\succ t[s]$. Why?

Consider an example.

$$\begin{array}{l} f(a) = a \\ f(f(a)) = a \\ f(f(f(a))) = a \end{array}$$

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.

Exercise: Show that $\{f(a) = a, f(f(f(a))) \neq a\}$ is unsatisfiable, by using superposition with redundancy elimination.

How to "come up" with simplification orderings?

Term Algebra

Term algebra $TA(\Sigma)$ of signature Σ:

- Domain: the set of all ground terms of Σ.
- Interpretation of any function symbol f or constant c is defined as follows::

$$\begin{array}{rcl} f_{TA(\Sigma)}(t_1, \ldots, t_n) & \overset{\text{def}}{\Leftrightarrow} & f(t_1, \ldots, t_n); \\ c_{TA(\Sigma)} & \overset{\text{def}}{\Leftrightarrow} & c. \end{array}$$

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \ldots, t_n) \succ_{KB} h(s_1, \ldots, s_m)$ if

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \dots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \dots, t_n) \succ_{KB} h(s_1, \dots, s_m)$ if

1. $|g(t_1, \dots, t_n)| > |h(s_1, \dots, s_m)|$ (by weight) or

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \ldots, t_n) \succ_{KB} h(s_1, \ldots, s_m)$ if

1. $|g(t_1, \ldots, t_n)| > |h(s_1, \ldots, s_m)|$ (by weight) or
2. $|g(t_1, \ldots, t_n)| = |h(s_1, \ldots, s_m)|$ and one of the following holds:
    - 2.1 $g \gg h$ (by precedence) or

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \dots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \dots, t_n) \succ_{KB} h(s_1, \dots, s_m)$ if

1. $|g(t_1, \dots, t_n)| > |h(s_1, \dots, s_m)|$ (by weight) or
2. $|g(t_1, \dots, t_n)| = |h(s_1, \dots, s_m)|$ and one of the following holds:
   - 2.1 $g \gg h$ (by precedence) or
   - 2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \dots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically).

Example

$$\begin{aligned} w(a) &= 1 \\ w(b) &= 2 \\ w(f) &= 3 \\ w(g) &= 0 \end{aligned}$$

$|f(g(a), f(a, b))|$

Example

$$\begin{array}{rcl} w(a) & = & 1 \\ w(b) & = & 2 \\ w(f) & = & 3 \\ w(g) & = & 0 \end{array}$$

$$|f(g(a), f(a, b))| = |3(0(1), 3(1, 2))|$$

Example

$$\begin{aligned} w(a) &= 1 \\ w(b) &= 2 \\ w(f) &= 3 \\ w(g) &= 0 \end{aligned}$$

$$|f(g(a), f(a, b))| = |3(0(1), 3(1, 2))| = 3 + 0 + 1 + 3 + 1 + 2$$

Example

$$\begin{array}{rcl} w(a) & = & 1 \\ w(b) & = & 2 \\ w(f) & = & 3 \\ w(g) & = & 0 \end{array}$$

$$|f(g(a), f(a, b))| = |3(0(1), 3(1, 2))| = 3 + 0 + 1 + 3 + 1 + 2 = 10.$$

Example

$$\begin{array}{rcl} w(a) & = & 1 \\ w(b) & = & 2 \\ w(f) & = & 3 \\ w(g) & = & 0 \end{array}$$

$$|f(g(a), f(a, b))| = |3(0(1), 3(1, 2))| = 3 + 0 + 1 + 3 + 1 + 2 = 10.$$

The Knuth-Bendix ordering is the main ordering used in Vampire and all other resolution and superposition theorem provers.

Knuth-Bendix Ordering (KBO), Ground Case: Summary

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \ldots, t_n) \succ_{KB} h(s_1, \ldots, s_m)$ if

1. $|g(t_1, \ldots, t_n)| > |h(s_1, \ldots, s_m)|$ (by weight) or
2. $|g(t_1, \ldots, t_n)| = |h(s_1, \ldots, s_m)|$ and one of the following holds:
   - 2.1 $g \gg h$ (by precedence) or
   - 2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \ldots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \rightarrow \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \ldots, t_n) \succ_{KB} h(s_1, \ldots, s_m)$ if

1. $|g(t_1, \ldots, t_n)| > |h(s_1, \ldots, s_m)|$ (by weight) or
2. $|g(t_1, \ldots, t_n)| = |h(s_1, \ldots, s_m)|$ and one of the following holds:
   - 2.1 $g \gg h$ (by precedence) or
   - 2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \ldots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Note: Weight functions w are **not arbitrary functions** – need to be “compatible” with $\gg$.

Knuth-Bendix Ordering (KBO), Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \dots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \dots, t_n) \succ_{KB} h(s_1, \dots, s_m)$ if

1. $|g(t_1, \dots, t_n)| > |h(s_1, \dots, s_m)|$ (by weight) or
2. $|g(t_1, \dots, t_n)| = |h(s_1, \dots, s_m)|$ and one of the following holds:
   - 2.1 $g \gg h$ (by precedence) or
   - 2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \dots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Note: Weight functions w are **not arbitrary functions** – need to be “compatible” with $\gg$.

Why? Compare for example a and $f(a)$ with arbitrary $\gg$ and w.

Weight Functions, Ground Case

A weight function $w : \Sigma \to \mathbb{N}$ is any function satisfying:

- $w(a) > 0$ for any constant $a \in \Sigma$;
- if $w(f) = 0$ for a unary function $f \in \Sigma$, then $f \gg g$ for all functions $g \in \Sigma$ with $f \neq g$.
  That is, f is the greatest element of Σ wrt $\gg$.

As a consequence, there is at most one unary function f with $w(f) = 0$.

Example

Consider the KBO ordering $\succ$ generated by the precedence $inverse \gg times$.

Consider the literal:

$$inverse(times(a, b)) = times(inverse(a), inverse(b)).$$

Compare, w.r.t $\succ$, the left- and right-hand side terms of the equality when:

- $weight(inverse) = weigth(times) = 1$;
- $weight(inverse) = 0$ and $weight(times) = 1$.

Example

Consider the KBO ordering $\succ$ generated by the precedence $inverse \gg times$.

Consider the literal:

$$inverse(times(a,b)) = times(inverse(a), inverse(b)).$$

Compare, w.r.t $\succ$, the left- and right-hand side terms of the equality when:

- $weight(inverse) = weigth(times) = 1$;
- $weight(inverse) = 0$ and $weight(times) = 1$.

Same Property for $\mathbb{S}\text{up}_{\succ,\sigma}$ as for $\mathbb{BR}_\sigma$

The conclusion is strictly smaller than the rightmost premise:

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

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.

New Redundancy

Consider a superposition with a unit left premise:

$$\frac{\underline{l = r} \qquad \underline{s[l] = t} \lor D}{s[r] = t \lor D}\ (\text{Sup}),$$

Note that we have

$$l = r, s[r] = t \lor D \models s[l] = t \lor D$$

New Redundancy

Consider a superposition with a unit left premise:

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

Note that we have

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

and we have

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

New Redundancy

Consider a superposition with a unit left premise:

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

Note that we have

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

and we have

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

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

New Redundancy

Consider a superposition with a unit left premise:

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

Note that we have

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

and we have

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

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

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

Exercise

Consider the KBO ordering $\succ$ generated by:

– the precedence $P \gg Q \gg f \gg a$;

and

– the weight function w with $w(P) = w(Q) = 2$, $w(f) = w(a) = 1$.

Consider the set of clauses S to be:

$$\begin{array}{l} Q(a), \\ \neg Q(a) \vee f(a) = a, \\ \neg P(a), \\ P(f(a))\}. \end{array}$$

Apply saturation on S by using an inferece process with redundancy based on the (ground) superposition calculus $\mathbb{S}\mathrm{up}_{\succ,\sigma}$.

Exercise

Consider the KBO ordering $\succ$ generated by:

– the precedence $f \gg a \gg b \gg c$;

and

– the weight function w with $w(f) = w(a) = w(b) = w(c) = 1$.

Consider the set S of ground formulas:

$$\begin{array}{l} a = b \vee a = c \\ f(a) \neq f(b) \\ b = c \end{array}$$

Apply saturation on S using an inference process based on the ground superposition calculus $\mathbb{S}\mathrm{up}_{\succ,\sigma}$ (including the inference rules of ground binary resolution with selection).

Show that S is unsatisfiable.

Exercise

Consider the KBO ordering $\succ$ generated by:

– the precedence $f \gg a \gg b \gg c$;

and

– the weight function w with $w(f) = w(a) = w(b) = w(c) = 1$.

Consider the set S of ground formulas:

$$\begin{array}{l} a = b \lor a = c \\ f(a) \neq f(b) \\ b = c \end{array}$$

Apply saturation on S using an inference process based on the ground superposition calculus $\mathbb{S}\mathrm{up}_{\succ,\sigma}$ (including the inference rules of ground binary resolution with selection).

Show that S is unsatisfiable.

Challenge: Show that S is unsatisfiable such that during saturation only 4 new clauses are generated.

Outline

Completeness of $\mathbb{S}\mathrm{up}_{\succ,\sigma}$

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

1. S_0 be a set of clauses;
2. $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ be a fair $\mathbb{S}\mathrm{up}_{\succ,\sigma}$-inference process with redundancy.

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

End of Lecture 3

Slides for lecture 3 ended here . . .

Outline

Setting the Scene

First-Order Theorem Proving - An Example

First-Order Logic and TPTP

Inference Systems

Selection Functions

Saturation Algorithms

Redundancy Elimination

Equality

Term Orderings

Completeness of Ground Superposition

Unification and Lifting

Non-Ground Superposition

Substitution

- A substitution θ is a mapping from variables to terms such that the set $\{x \mid \theta(x) \neq x\}$ is finite.
- This set is called the domain of θ.
- Notation: $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $x_1, \ldots, x_n$ are pairwise different variables, denotes the substitution θ such that

$$\theta(x) = \begin{cases} t_i & \text{if } x = x_i; \\ x & \text{if } x \notin \{x_1, \ldots, x_n\}. \end{cases}$$

- Application of this substitution to an expression E: simultaneous replacement of x_i by t_i.
- Application of a substitution θ to E is denoted by $E\theta$.
- Since substitutions are functions, we can define their composition (written $\sigma\tau$ instead of $\tau \circ \sigma$). Note that we have $E(\sigma\tau) = (E\sigma)\tau$.

Substitution

- A substitution θ is a mapping from variables to terms such that the set $\{x \mid \theta(x) \neq x\}$ is finite.
- This set is called the domain of θ.
- Notation: $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $x_1, \ldots, x_n$ are pairwise different variables, denotes the substitution θ such that

$$\theta(x) = \begin{cases} t_i & \text{if } x = x_i; \\ x & \text{if } x \notin \{x_1, \ldots, x_n\}. \end{cases}$$

- Application of this substitution to an expression E: simultaneous replacement of x_i by t_i.
- Application of a substitution θ to E is denoted by $E\theta$.
- Since substitutions are functions, we can define their composition (written $\sigma\tau$ instead of $\tau \circ \sigma$). Note that we have $E(\sigma\tau) = (E\sigma)\tau$.

Example

Consider:

$$E = p(x, y, f(a))$$
$$\theta = \{x \mapsto b, y \mapsto x\}$$

What is $E\theta$?

Substitution composition

Suppose we have two substitutions

$$\theta_1 = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\} \text{ and}$$
$$\theta_2 = \{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}.$$

How can we compute their composition $\theta_1\theta_2$?

Substitution composition

Suppose we have two substitutions

$$\theta_1 = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\} \text{ and}$$
$$\theta_2 = \{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}.$$

How can we compute their composition $\theta_1\theta_2$?

The substitution $\theta_1\theta_2$ is obtained from the set:

$$\{x_1 \mapsto s_1\theta_2, \ldots, x_m \mapsto s_m\theta_2,$$
$$y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\},$$

Substitution composition

Suppose we have two substitutions

$$\theta_1 = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\} \text{ and}$$
$$\theta_2 = \{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}.$$

How can we compute their composition $\theta_1\theta_2$?

The substitution $\theta_1\theta_2$ is obtained from the set:

$$\{x_1 \mapsto s_1\theta_2, \ldots, x_m \mapsto s_m\theta_2,$$
$$y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\},$$

by deleting

- all $y_i \mapsto t_i$ with $y_i \in \{x_1, \ldots, x_m\}$,
- all $x_i \mapsto s_i\theta_2$ with $x_i = s_i\theta_2$.

Example

Consider:

$$\begin{aligned}\theta_1 &= \{x \mapsto f(y), y \mapsto z\},\\ \theta_2 &= \{x \mapsto a, y \mapsto b, z \mapsto y\}.\end{aligned}$$

What is $\theta_1\theta_2$?

Instances, Ground

An instance of an expression (that is term, atom, literal, or clause) E is obtained by applying a substitution to E. Examples:

- some instances of the term $f(x, a, g(x))$ are:
  $f(x, a, g(x))$,
  $f(y, a, g(y))$,
  $f(a, a, g(a))$,
  $f(g(b), a, g(g(b)))$;
- but the term $f(b, a, g(c))$ is not an instance of this term.

Ground instance: instance with no variables.

Herbrand's Theorem

For a set of clauses S denote by S^* the set of ground instances of clauses in S.

Theorem Let Σ be a signature with at least one constant symbol and S be a set of (universal) clauses over Σ. The following conditions are equivalent.

1. S is unsatisfiable;
2. S^* is unsatisfiable;

Herbrand's Theorem

For a set of clauses S denote by S^* the set of ground instances of clauses in S.

Theorem Let Σ be a signature with at least one constant symbol and S be a set of (universal) clauses over Σ. The following conditions are equivalent.

1. S is unsatisfiable;
2. S^* is unsatisfiable;

By compactness of first-order logic the last condition is equivalent to

3. there exists a finite unsatisfiable set of ground instances of clauses in S.

Herbrand's Theorem

For a set of clauses S denote by S^* the set of ground instances of clauses in S.

Theorem Let Σ be a signature with at least one constant symbol and S be a set of (universal) clauses over Σ. The following conditions are equivalent.

1. S is unsatisfiable;
2. S^* is unsatisfiable;

By compactness of first-order logic the last condition is equivalent to

3. there exists a finite unsatisfiable set of ground instances of clauses in S.

The theorem reduces the problem of checking unsatisfiability of sets of arbitrary clauses to checking unsatisfiability of sets of ground clauses . . .

Herbrand's Theorem

For a set of clauses S denote by S^* the set of ground instances of clauses in S.

Theorem Let Σ be a signature with at least one constant symbol and S be a set of (universal) clauses over Σ. The following conditions are equivalent.

1. S is unsatisfiable;
2. S^* is unsatisfiable;

By compactness of first-order logic the last condition is equivalent to

3. there exists a finite unsatisfiable set of ground instances of clauses in S.

The theorem reduces the problem of checking unsatisfiability of sets of arbitrary clauses to checking unsatisfiability of sets of ground clauses . . .

The only problem is that S^* can be infinite even if S is finite.

Lifting

Lifting is a technique for proving completeness theorems in the following way:

1. Prove completeness of the system for a set of ground clauses;
2. Lift the proof to the non-ground case.

Lifting, Example

Consider two (non-ground) clauses $p(x, a) \vee q_1(x)$ and $\neg p(y, z) \vee q_2(y, z)$. If the signature contains function symbols, then both clauses have infinite sets of instances:

$$\begin{array}{rcl} \{p(r, a) \vee q_1(r) & \mid & r \text{ is ground}\} \\ \{\neg p(s, t) \vee q_2(s, t) & \mid & s, t \text{ are ground}\} \end{array}$$

We can resolve such instances if and only if $r = s$ and $t = a$. Then we can apply the following inference

$$\frac{p(s, a) \vee q_1(s) \quad \neg p(s, a) \vee q_2(s, a)}{q_1(s) \vee q_2(s, a)} \text{ (BR)}$$

But there is an infinite number of such inferences.

Lifting, Idea

The idea is to represent an infinite number of ground inferences of the form

$$\frac{p(s,a) \vee q_1(s) \quad \neg p(s,a) \vee q_2(s,a)}{q_1(s) \vee q_2(s,a)} \text{ (BR)}$$

by a single non-ground inference

$$\frac{p(x,a) \vee q_1(x) \quad \neg p(y,z) \vee q_2(y,z)}{q_1(y) \vee q_2(y,a)} \text{ (BR)}$$

Is this always possible?

Yes!

$$\frac{p(x,a) \lor q_1(x) \quad \neg p(y,z) \lor q_2(y,z)}{q_1(y) \lor q_2(y,a)} \text{ (BR)}$$

Note that the substitution $\{x \mapsto y, z \mapsto a\}$ is a solution of the "equation" $p(x,a) = p(y,z)$.

Lifting

Idea: Represent an infinite number of ground inferences by a single non-ground inference.

Lifting (Robinson, 1965)

Idea: Represent an infinite number of ground inferences by a single non-ground inference.

In case of $\mathbb{BR}$:

- Resolution for non-ground clauses
- The notion of "same" ground atoms is generalized to unifiability of non-ground atoms;
- Only compute substitutions that are most general unifiers (mgu).

Lifting (Robinson, 1965)

Lifting Lemma for BR in $\mathbb{BR}$:

Let C and D clauses with no shared variables. If:

$$\begin{tikzcd} C \arrow[d, "\sigma_1"] & D \arrow[d, "\sigma_2"] \\ C\sigma_1 & D\sigma_2 \end{tikzcd}$$

$$\frac{C\sigma_1 \qquad D\sigma_2}{C'} \ (\textit{ground}\ \mathrm{BR})$$

then there exists a substitution σ sucht that:

$$\frac{C \quad D}{C''} \ (\textit{non} - \textit{ground}\ \mathrm{BR})$$

$$\begin{tikzcd} C'' \arrow[d, "\sigma"] \\ C' = C''\sigma \end{tikzcd}$$

Lifting (Robinson, 1965)(Bachmair & Ganzinger, 1990)

Lifting Lemma for BR in $\mathbb{BR}$:

Let C and D clauses with no shared variables. If:

$$\begin{array}{c} \begin{array}{cc} C & D \\ \big\downarrow\sigma_1 & \big\downarrow\sigma_2 \end{array} \\ \dfrac{C\sigma_1 \qquad D\sigma_2}{C'} \end{array} \; (\mathit{ground}\ \mathrm{BR})$$

then there exists a substitution σ sucht that:

$$\begin{array}{c} \dfrac{C \quad D}{C''} \\ \big\downarrow\sigma \\ C' = C''\sigma \end{array} \; (\mathit{non} - \mathit{ground}\ \mathrm{BR})$$

Similar lifting lemmas each inferences of $\mathbb{BR}$ and $\mathbb{SRF}$.

What should we lift?

- Ordering $\succ$;
- Selection function σ;
- Calculus $\mathbb{S}\mathrm{up}_{\succ,\sigma}$.

Most importantly, for the lifting to work we should be able to solve equations $s = t$ between terms and between atoms. This can be done using most general unifiers.

Unifier

Unifier of expressions s_1 and s_2: a substitution θ such that $s_1\theta = s_2\theta$. In other words, a unifier is a solution to an "equation" $s_1 = s_2$. In a similar way we can define solutions to systems of equations $s_1 = s'_1, \dots, s_n = s'_n$. We call such solutions simultaneous unifiers of $s_1, \dots, s_n$ and $s'_1, \dots, s'_n$.

(Most General) Unifiers

A solution θ to a set of equations E is said to be a most general solution if for every other solution σ there exists a substitution τ such that $\theta\tau = \sigma$. In a similar way can define a most general unifier.

(Most General) Unifiers

A solution θ to a set of equations E is said to be a most general solution if for every other solution σ there exists a substitution τ such that $\theta\tau = \sigma$. In a similar way can define a most general unifier.

Consider terms $f(x_1, g(x_1), x_2)$ and $f(y_1, y_2, y_2)$.
(Some of) their unifiers are
$\theta_1 = \{y_1 \mapsto x_1, y_2 \mapsto g(x_1), x_2 \mapsto g(x_1)\}$ and
$\theta_2 = \{y_1 \mapsto a, y_2 \mapsto g(a), x_2 \mapsto g(a), x_1 \mapsto a\}$:

$f(x_1, g(x_1), x_2)\theta_1 = f(x_1, g(x_1), g(x_1))$;
$f(y_1, y_2, y_2)\theta_1 = f(x_1, g(x_1), g(x_1))$;
$f(x_1, g(x_1), x_2)\theta_2 = f(a, g(a), g(a))$;
$f(y_1, y_2, y_2)\theta_2 = f(a, g(a), g(a))$.

But only θ_1 is most general.

Unification

Let E be a set of equations. An isolated equation in E is any equation $x = t$ in E such that x has exactly one occurrence in E.

```
input:
    A finite set of equations $E$
    ($s, t$ denote terms, $c, d$ constants, $f, g$ function symbols, $x$ variable)
output:
    A solution to $E$ or failure.
begin
    while there exists a non-isolated equation $(s = t) \in E$
    do
        case $(s, t)$ of
            $(t, t) \Rightarrow$ Remove this equation from $E$
            $(x, t) \Rightarrow$
                if $x$ occurs in $t$
                then halt with failure
                else replace $x$ by $t$ in all other equations of $E$
            $(t, x) \Rightarrow$ replace this equation by $x = t$
                      and do the same as in the case $(x, t)$
            $(c, d) \Rightarrow$ halt with failure
            $(c, f(t_1, \ldots, t_n)) \Rightarrow$ halt with failure
            $(f(t_1, \ldots, t_n), c) \Rightarrow$ halt with failure
            $(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n)) \Rightarrow$ halt with failure
            $(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) \Rightarrow$ replace this equation by the set
                      $s_1 = t_1, \ldots, s_n = t_n$
            end
        od
        Now $E$ has the form $\{x_1 = r_1, \ldots, x_l = r_l\}$ and every equation in it
        is isolated
        return the substitution $\{x_1 \mapsto r_1, \ldots, x_l \mapsto r_l\}$
    end
```

Examples

$$\{h(g(f(x), a)) = h(g(y, y))\}$$
$$\{h(f(y), y, f(z)) = h(z, f(x), x)\}$$
$$\{h(g(f(x), z)) = h(g(y, y))\}$$

Properties

Theorem Suppose we run the unification algorithm on $s = t$. Then

- If s and t are unifiable, then the algorithms terminates and outputs a most general unifier of s and t.
- If s and t are not unifiable, then the algorithms terminates with failure.

Notation (slightly ambiguous):

- $mgu(s, t)$ for a most general unifier;
- $mgs(E)$ for a most general solution.

Outline

Revisit: What should we lift?

- Ordering $\succ$;
- Selection function σ;
- Calculus $\mathbb{S}\mathrm{up}_{\succ,\sigma}$ (thanks to lifting lemmas).

Most importantly, for the lifting to work we use most general unifiers.

Knuth-Bendix Ordering (KBO), Ground Case (Recap)

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \ldots, t_n) \succ_{KB} h(s_1, \ldots, s_m)$ if

1. $|g(t_1, \ldots, t_n)| > |h(s_1, \ldots, s_m)|$ (by weight) or
2. $|g(t_1, \ldots, t_n)| = |h(s_1, \ldots, s_m)|$ and one of the following holds:
   2.1 $g \gg h$ (by precedence) or
   2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \ldots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Knuth-Bendix Ordering (KBO), Ground Case (Recap)

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \dots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \dots, t_n) \succ_{KB} h(s_1, \dots, s_m)$ if

1. $|g(t_1, \dots, t_n)| > |h(s_1, \dots, s_m)|$ (by weight) or
2. $|g(t_1, \dots, t_n)| = |h(s_1, \dots, s_m)|$ and one of the following holds:
   - 2.1 $g \gg h$ (by precedence) or
   - 2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \dots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Note: Weight functions w are **not arbitrary functions**
– need to be "compatible" with $\gg$.

Knuth-Bendix Ordering (KBO), Ground Case (Recap)

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \to \mathbb{N}$.

Weight of a ground term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$g(t_1, \ldots, t_n) \succ_{KB} h(s_1, \ldots, s_m)$ if

1. $|g(t_1, \ldots, t_n)| > |h(s_1, \ldots, s_m)|$ (by weight) or
2. $|g(t_1, \ldots, t_n)| = |h(s_1, \ldots, s_m)|$ and one of the following holds:
   - 2.1 $g \gg h$ (by precedence) or
   - 2.2 $g = h$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \ldots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Note: Weight functions w are **not arbitrary functions**
– need to be “compatible” with $\gg$.

Why? Compare for example a and $f(a)$ with arbitrary $\gg$ and w.

Weight Functions, Ground Case

A weight function $w : \Sigma \to \mathbb{N}$ is any function satisfying:

- $w(a) > 0$ for any constant $a \in \Sigma$;
- if $w(f) = 0$ for a unary function $f \in \Sigma$, then $f \gg g$ for all functions $g \in \Sigma$ with $f \neq g$.
  That is, f is the greatest element of Σ wrt $\gg$.

As a consequence, there is at most one unary function f with $w(f) = 0$.

Weight Functions, Non-Ground Case

A weight function $w : \Sigma \cup \mathit{Vars} \to \mathbb{N}$, with Vars denoting the set of variables, is any function satisfying:

- $w(x) = v_0$ for all variables $x \in \mathit{Vars}$, where $v_0 > 0$;
- $w(a) \geq v_0$ for any constant $a \in \Sigma$;
- if $w(f) = 0$ for a unary function $f \in \Sigma$, then $f \gg g$ for all functions $g \in \Sigma$ with $f \neq g$.
  That is, f is the greatest element of Σ wrt $\gg$.

As a consequence, there is at most one unary function f with $w(f) = 0$.

Weight Functions, Non-Ground Case

A weight function $w : \Sigma \cup \mathit{Vars} \rightarrow \mathbb{N}$, with Vars denoting the set of variables, is any function satisfying:

- $w(x) = v_0$ for all variables $x \in \mathit{Vars}$, where $v_0 > 0$;
- $w(a) \geq v_0$ for any constant $a \in \Sigma$;
- if $w(f) = 0$ for a unary function $f \in \Sigma$, then $f \gg g$ for all functions $g \in \Sigma$ with $f \neq g$.
  That is, f is the greatest element of Σ wrt $\gg$.

As a consequence, there is at most one unary function f with $w(f) = 0$.

Notation: Given a term s and variable x, we write $\#(x, s)$ to denote the number of occurences of x in s.

Knuth-Bendix Ordering (KBO), Non-Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \cup Vars \rightarrow \mathbb{N}$.

Weight of a term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

Knuth-Bendix Ordering (KBO), Non-Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \cup Vars \to \mathbb{N}$.

Weight of a term t is

$$|g(t_1, \dots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$s \succ_{KB} t$ if

1. $\#(x, s) \geq \#(x, t)$ for all variables x and $|s| > |t|$ (by weight) or
2. $\#(x, s) \geq \#(x, t)$ for all variables x and $|s| = |t|$ and one of the following holds:
   - 2.1 $t = x$, $s = f^n(x)$ for some $n \geq 1$, or
   - 2.2 $s = g(t_1, \dots, t_n)$, $t = h(s_1, \dots, s_m)$ and $g \gg h$ (by precedence) or
   - 2.3 $s = g(t_1, \dots, t_n)$, $t = g(s_1, \dots, s_n)$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \dots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Knuth-Bendix Ordering (KBO), Non-Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \cup Vars \to \mathbb{N}$.

Weight of a term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$s \succ_{KB} t$ if

1. $\#(x, s) \geq \#(x, t)$ for all variables x and $|s| > |t|$ (by weight) or
2. $\#(x, s) \geq \#(x, t)$ for all variables x and $|s| = |t|$ and one of the following holds:
    - 2.1 $t = x$, $s = f^n(x)$ for some $n \geq 1$, or
    - 2.2 $s = g(t_1, \ldots, t_n)$, $t = h(s_1, \ldots, s_m)$ and $g \gg h$ (by precedence) or
    - 2.3 $s = g(t_1, \ldots, t_n)$, $t = g(s_1, \ldots, s_n)$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \ldots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Knuth-Bendix Ordering (KBO), Non-Ground Case

Let us fix

- Signature Σ, it induces the term algebra $TA(\Sigma)$.
- Total ordering $\gg$ on Σ, called precedence relation;
- Weight function $w : \Sigma \cup Vars \to \mathbb{N}$.

Weight of a term t is

$$|g(t_1, \ldots, t_n)| = w(g) + \sum_{i=1}^{n} |t_i|.$$

$s \succ_{KB} t$ if

1. $\#(x, s) \geq \#(x, t)$ for all variables x and $|s| > |t|$ (by weight) or
2. $\#(x, s) \geq \#(x, t)$ for all variables x and $|s| = |t|$ and one of the following holds:
   - 2.1 $t = x$, $s = f^n(x)$ for some $n \geq 1$, or
   - 2.2 $s = g(t_1, \ldots, t_n)$, $t = h(s_1, \ldots, s_m)$ and $g \gg h$ (by precedence) or
   - 2.3 $s = g(t_1, \ldots, t_n)$, $t = g(s_1, \ldots, s_n)$ and for some $1 \leq i \leq n$ we have $t_1 = s_1, \ldots, t_{i-1} = s_{i-1}$ and $t_i \succ_{KB} s_i$ (lexicographically, i.e. left-to-right).

Selection Functions, Lifting

If for some grounding substitution θ, $L\theta$ is selected in $L\theta \vee C\theta$, then L is selected in $L \vee C$.

If the ground selection function is well-behaved, then its corresponding non-ground selection function lifted as above is also well-behaved.

Selection Functions, Lifting

If for some grounding substitution θ, $L\theta$ is selected in $L\theta \vee C\theta$, then L is selected in $L \vee C$.

If the ground selection function is well-behaved, then its corresponding non-ground selection function lifted as above is also well-behaved.

Non-Ground Superposition, Lifting

Superposition:

$$\frac{\underline{l = r} \vee C \quad \underline{s[l'] = t} \vee D}{(s[r] = t \vee C \vee D)\theta}\ (Sup), \qquad \frac{\underline{l = r} \vee C \quad \underline{s[l'] \neq t} \vee D}{(s[r] \neq t \vee C \vee D)\theta}\ (Sup),$$

where

1. θ is an mgu of l and l';
2. l' is not a variable;
3. $r\theta \not\succeq l\theta$;
4. $t\theta \not\succeq s[l']\theta$.

Non-Ground Superposition, Lifting

Superposition:

$$\frac{\underline{l = r} \vee C \quad \underline{s[l'] = t} \vee D}{(s[r] = t \vee C \vee D)\theta}\ (Sup), \qquad \frac{\underline{l = r} \vee C \quad \underline{s[l'] \neq t} \vee D}{(s[r] \neq t \vee C \vee D)\theta}\ (Sup),$$

where

1. θ is an mgu of l and l';
2. l' is not a variable;
3. $r\theta \not\succeq l\theta$;
4. $t\theta \not\succeq s[l']\theta$.

Observations:

- ordering is partial, hence conditions like $r\theta \not\succeq l\theta$;
- these conditions must be checked a posteriori, that is, after the rule has been applied.

Note, however, that $l \succ r$ implies $l\theta \succ r\theta$, so checking orderings a priory helps.

More rules

Equality Resolution:

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

where θ is an mgu of s and s'.

Equality Factoring:

$$\frac{\underline{l = r} \vee l' = r' \vee C}{(l = r \vee r \neq r' \vee C)\theta}\ \text{(EF)},$$

where θ is an mgu of l and l', $r\theta \not\succeq l\theta$, $r'\theta \not\succeq l\theta$, and $r'\theta \not\succeq r\theta$.

Non-Ground Binary Resolution

- Binary resolution,

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

  where θ is the mgu of P and P'.

- Positive factoring,

$$\frac{\underline{P} \vee \underline{P'} \vee C}{(P \vee C)\theta} \text{ (Fact)}.$$

  where θ is the mgu of P and P'.

- Negative factoring,

$$\frac{\underline{\neg P} \vee \underline{\neg P'} \vee C}{(\neg P \vee C)\theta} \text{ (Fact)}.$$

  where θ is the mgu of P and P'.

Checking Redundancy

Suppose that the current search space S contains no redundant clauses. How can a redundant clause appear in the inference process?

Checking Redundancy

Suppose that the current search space S contains no redundant clauses. How can a redundant clause appear in the inference process?
Only when a new clause (a child of the selected clause and possibly other clauses) is added.
Classification of redundancy checks:

- The child is redundant;
- The child makes one of the clauses in the search space redundant.

Checking Redundancy

Suppose that the current search space S contains no redundant clauses. How can a redundant clause appear in the inference process?
Only when a new clause (a child of the selected clause and possibly other clauses) is added.
Classification of redundancy checks:

- The child is redundant;
- The child makes one of the clauses in the search space redundant.

We use some fair strategy and perform these checks after every inference that generates a new clause.
In fact, one can do better in some of the cases.

Subsumption, Non-Ground Case

A clause C subsumes any clause D if $C\theta \subseteq D$ for some substitution θ.

Subsumption, Non-Ground Case

A clause C subsumes any clause D if $C\theta \subseteq D$ for some substitution θ.

Subsumption and redundancy: If a clause set S contains two different clauses C and D and C subsumes D, then D is redundant in S (and can be removed).

Demodulation, Non-Ground Case

$$\frac{l = r \quad \cancel{L[l'] \vee D}}{L[r\theta] \vee D}\ (\text{Dem}),$$

where $l\theta = l'$, $l\theta \succ r\theta$, and $(L[l'] \vee D) \succ (l\theta = r\theta)$.

Demodulation, Non-Ground Case

$$\frac{l = r \quad \cancel{L[l'] \vee D}}{L[r\theta] \vee D} \ \text{(Dem)},$$

where $l\theta = l'$, $l\theta \succ r\theta$, and $(L[l'] \vee D) \succ (l\theta = r\theta)$.

Easier to understand:

$$\frac{l = r \quad \cancel{L[l\theta] \vee D}}{L[r\theta] \vee D} \ \text{(Dem)},$$

where $l\theta \succ r\theta$, and $(L[l\theta] \vee D) \succ (l\theta = r\theta)$.

General Redundancy, Non-Ground Case

D is redundant wrt C if D^* is redundant wrt C^*,
where D^* and C^* are respectively the set of ground instances of D and C.

Consider two non-ground clauses C, D.

To show that D is redundant wrt C, it is sufficient to find a substitution θ such that:

1. $D^* \succ C\theta$;
2. D^* is a logical consequence of $C\theta$,

for *any* ground instance D^* of D.

General Redundancy, Non-Ground Case

D is redundant wrt C if D^* is redundant wrt C^*,
where D^* and C^* are respectively the set of ground instances of D and C.

Consider two non-ground clauses C, D.

To show that D is redundant wrt C, it is sufficient to find a substitution θ such that:

1. $D^* \succ C\theta$;
2. D^* is a logical consequence of $C\theta$,

for *any* ground instance D^* of D.

Generating and Simplifying Inferences

An inference

$$\frac{C_1 \quad \ldots \quad C_n}{C} \; .$$

is called simplifying if at least one premise C_i becomes redundant after the addition of the conclusion C to the search space. We then say that C_i is simplified into C.
A non-simplifying inference is called generating.

Generating and Simplifying Inferences

An inference

$$\frac{C_1 \quad \ldots \quad C_n}{C} \ .$$

is called simplifying if at least one premise C_i becomes redundant after the addition of the conclusion C to the search space. We then say that C_i is simplified into C.
A non-simplifying inference is called generating.

Note. The property of being simplifying is undecidable. So is the property of being redundant. So in practice we employ sufficient conditions for simplifying inferences and for redundancy.

Generating and Simplifying Inferences

An inference

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

is called simplifying if at least one premise C_i becomes redundant after the addition of the conclusion C to the search space. We then say that C_i is simplified into C.
A non-simplifying inference is called generating.

Note. The property of being simplifying is undecidable. So is the property of being redundant. So in practice we employ sufficient conditions for simplifying inferences and for redundancy.

Idea: try to search eagerly for simplifying inferences bypassing the strategy for inference selection.

Generating and Simplifying Inferences

Two main implementation principles:

apply simplifying inferences eagerly;
apply generating inferences lazily.

checking for simplifying inferences should pay off;
so it must be cheap.

Redundancy Checking

Redundancy-checking occurs upon addition of a new child C. It works as follows

- Retention test: check if C is redundant.
- Forward simplification: check if C can be simplified using a simplifying inference.
- Backward simplification: check if C simplifies or makes redundant an old clause.

Examples

Retention test:

- tautology-check;
- subsumption.

Simplification:

- demodulation (forward and backward);
- subsumption resolution (forward and backward):

$$\frac{A \vee C \quad \cancel{\neg B \vee D}}{D}\ (\text{Subs}), \qquad \text{or} \qquad \frac{\neg A \vee C \quad \cancel{B \vee D}}{D}\ (\text{Subs}),$$

such that for some substitution θ we have $A\theta \vee C\theta \subseteq B \vee D$.

Some redundancy criteria are expensive

- Tautology-checking is based on congruence closure.
- Subsumption and subsumption resolution are NP-complete.

Observations

- There may be chains (repeated applications) of forward simplifications.
- After a chain of forward simplifications another retention test can (should) be done.

Observations

- There may be chains (repeated applications) of forward simplifications.
- After a chain of forward simplifications another retention test can (should) be done.
- Backward simplification is often expensive.

Observations

- There may be chains (repeated applications) of forward simplifications.
- After a chain of forward simplifications another retention test can (should) be done.
- Backward simplification is often expensive.
- In practice, the retention test may include other checks, resulting in the loss of completeness, for example, we may decide to discard too heavy clauses.