Automated Deduction

Laura Kovács

TU Wien

Outline

First-Order Logic (recap) and TPTP

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

First-Order Logic (recap) and TPTP

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

First-Order Logic (recap) and TPTP

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

First-Order Logic (recap) and TPTP

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

First-Order Logic (recap) and TPTP

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

First-Order Logic (recap) and TPTP

- Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.
  In TPTP: Variable names start with upper-case letters.
- Terms: variables, constants, and expressions $f(t_1, \ldots, t_n)$, 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 (recap) and TPTP

- Language: variables, function and predicate (relation) symbols. A constant symbol is a special case of a function symbol.
  In TPTP: Variable names start with upper-case letters.
- Terms: variables, constants, and expressions $f(t_1, \ldots, t_n)$, 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 $>$ 45 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

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 $\cdot$.

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

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

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

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, $2 \cdot 2 = 4$).

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

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 $|\varepsilon + \varepsilon = |\varepsilon$.

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

Arbitrary First-Order Formulas (recap)

- 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 (recap)

- 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 (recap)

- 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 $\square$: clause with 0 literals, that is, when $n = 0$.

Clauses (recap)

- 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 $\square$: clause with 0 literals, that is, when $n = 0$.
- A formula in Clausal Normal Form (CNF): a conjunction of clauses.

Clauses (recap)

- 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 $\square$: 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:

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

- Factoring, denoted by Fact:

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

Soundness

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

Soundness

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

$\mathbb{BR}$ is sound.

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

Example

Consider the following set of clauses

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

The following derivation derives the empty clause from this set:

$$\dfrac{\dfrac{p \vee q \quad p \vee \neg q}{p \vee p}\,(\text{BR})}{p}\,(\text{Fact}) \qquad \dfrac{\dfrac{\neg p \vee q \quad \neg p \vee \neg q}{\neg p \vee \neg p}\,(\text{BR})}{\neg p}\,(\text{Fact})$$
$$\dfrac{}{\square}\,(\text{BR})$$

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

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

Binary Resolution with Selection

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

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

Completeness?

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

Consider this set of clauses:

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

It is unsatisfiable:

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

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

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

Literal Orderings

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

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

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

A well-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.

Outline

How to Establish Unsatisfiability?

Completess 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?

Completess 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 F is derivable in $\mathbb{I}$ from S_0. In particular, every S_i is a subset of the $\mathbb{I}$-closure of S_0.