DTU Course 02156 Logical Systems and Logic Programming (2021)

Week	Date	Main Topics (Prolog Programming in All Lessons)
35 #01	31/8	Course Prerequisites & Tutorial on Logical Systems and Logic Programming
36 #02	7/9	Chapter 1 - Introduction (Prolog Note)
37 #03	14/9	Chapter 2 - Propositional Logic: Formulas, Models, Tableaux
38 #04	21/9	Chapter 3 - Propositional Logic: Deductive Systems
39 #05	28/9	"Isabelle" - Propositional Logic: Sequent Calculus Verifier (SeCaV)
40 #06	5/10	Chapter 4 - Propositional Logic: Resolution
41 #07	12/10	Chapter 7 - First-Order Logic: Formulas, Models, Tableaux
42		(Autumn Vacation)
43 #08	26/10	Chapter 8 - First-Order Logic: Deductive Systems
44 #09	2/11	"Isabelle" - First-Order Logic: Sequent Calculus Verifier (SeCaV)
45 #10	9/11	Chapter 9 - First-Order Logic: Terms and Normal Forms
46 #11	16/11	Chapter 10 - First-Order Logic: Resolution
47 #12	23/11	Chapter 11 - First-Order Logic: Logic Programming
48 #13	30/11	Chapter 12 - First-Order Logic: Undecidability and Model Theory & Course Evaluation

Responsible: Associate Professor Jørgen Villadsen <jovi@dtu.dk>

Assignments & Exam MUST BE SOLVED INDIVIDUALLY

Assignment-1 Deadline Sunday 26/9 (Available Wednesday 15/9)

Assignment-2 Deadline Sunday 10/10 (Available Wednesday 29/9)

Assignment-3 Deadline Sunday 31/10 (Available Wednesday 13/10)

Assignment-4 Deadline Sunday 14/11 (Available Wednesday 3/11)

Assignment-5 Deadline Thursday 2/12 (Available Wednesday 17/11)

Written Exam Tuesday 14/12 (2 Hours / No Computer / All Notes Allowed)

The mandatory assignments and the written exam are evaluated as a whole – even if you do well in the mandatory assignments then you still must do decent in the written exam in order to pass the course!

A TEACHER MUST IMMEDIATELY REPORT ANY SUSPICION OF CHEATING TO THE STUDY ADMINISTRATION FOR FURTHER ACTIONS

Agenda — Week #11

Test

Prolog note — `copy_term`

Theorem Provers

Unification

Resolution

Test

1. Is the formula $\forall x((p \vee q(x)) \wedge r(a))$ a skolemization of the formula $(p \vee \forall x q(x)) \wedge \exists x r(x)$?

2. Is the formula $p \wedge \neg(q \vee r)$ in CNF (Conjunctive Normal Form)?

3. Is the formula $\forall x(p(x) \wedge q(a))$ a skolemization of the formula $\forall x p(x) \wedge \exists x q(x)$?

4. Is the formula $(p \vee q) \wedge \neg r$ in CNF (Conjunctive Normal Form)?

5. Is the formula $\forall x(p(x) \wedge q(f(x)))$ a skolemization of the formula $\forall x p(x) \wedge \exists x q(x)$?

6. Is $(\{1, 2, 3\}, \{\{\}\}, \{\}, \{\})$ a model for the formula $\neg\exists x p(x)$?

Renaming of Variables in Terms — Example

Consider the following queries:

```
?- X = p(A), X = Y, Y =.. [P,B], A = a.

X = p(a)
A = a
Y = p(a)
P = p
B = a

Yes

?- X = p(A), X = Y, Y =.. [P,B], A = a, B = b.

No
```

Renaming of Variables in Terms — Definition

There is a special predicate for such quite rare situations:

`copy_term(?Term1,?Term2)` succeeds iff `Term2` unifies with a renamed copy of `Term1`.

For example:

```
?- X = p(A), copy_term(X,Y), Y =.. [P,B], A = a, B = b.

X = p(a)
A = a
Y = p(b)
P = p
B = b

Yes
```

Renaming of Variables in Terms — Implementation

Note that copy_term/2 can be defined as follows:

```
copy_term(Term1,Term2) :-
  asserta(copy_term(Term1)), retract(copy_term(Term2)).
```

It is assumed that copy_term/1 is not used elsewhere (however this is not a strict requirement).

Otter — Wikipedia & Manual

Otter . . . was the first widely distributed high-performance theorem prover for first-order logic, and pioneered a number of important implementation techniques.

Otter — Wikipedia & Manual

Otter . . . was the first widely distributed high-performance theorem prover for first-order logic, and pioneered a number of important implementation techniques.

Otter is an acronym for Organized Techniques for Theorem proving and Efficient Research.

Otter — Wikipedia & Manual

Otter . . . was the first widely distributed high-performance theorem prover for first-order logic, and pioneered a number of important implementation techniques.

Otter is an acronym for Organized Techniques for Theorem proving and Efficient Research.

Otter is a resolution-style theorem prover for first-order logic with equality.

Otter — Wikipedia & Manual

Otter . . . was the first widely distributed high-performance theorem prover for first-order logic, and pioneered a number of important implementation techniques.

Otter is an acronym for Organized Techniques for Theorem proving and Efficient Research.

Otter is a resolution-style theorem prover for first-order logic with equality.

Otter includes the inference rules binary resolution, hyperresolution, . . .

Otter — Wikipedia & Manual

Otter . . . was the first widely distributed high-performance theorem prover for first-order logic, and pioneered a number of important implementation techniques.

Otter is an acronym for Organized Techniques for Theorem proving and Efficient Research.

Otter is a resolution-style theorem prover for first-order logic with equality.

Otter includes the inference rules binary resolution, hyperresolution, . . .

Some of its other abilities and features are conversion from first-order formulas to clauses, . . . factoring, . . .

Otter — Wikipedia & Manual

Otter . . . was the first widely distributed high-performance theorem prover for first-order logic, and pioneered a number of important implementation techniques.

Otter is an acronym for Organized Techniques for Theorem proving and Efficient Research.

Otter is a resolution-style theorem prover for first-order logic with equality.

Otter includes the inference rules binary resolution, hyperresolution, . . .

Some of its other abilities and features are conversion from first-order formulas to clauses, . . . factoring, . . .

Otter is coded in ANSI C, is free, and is portable to many different kinds of computers.

Prolog: leanTaP & leanCoP

leanTaP is written in Prolog and implements a theorem prover for first-order logic based on free-variable tableaux.

Prolog: leanTaP & leanCoP

leanTaP is written in Prolog and implements a theorem prover for first-order logic based on free-variable tableaux.

The unique thing about leanTaP is its small size.

Prolog: leanTaP & leanCoP

leanTaP is written in Prolog and implements a theorem prover for first-order logic based on free-variable tableaux.

The unique thing about leanTaP is its small size.

The minimal version of the Prolog source code is only 360 bytes.

Prolog: leanTaP & leanCoP

leanTaP is written in Prolog and implements a theorem prover for first-order logic based on free-variable tableaux.

The unique thing about leanTaP is its small size.

The minimal version of the Prolog source code is only 360 bytes.

But recently 199 bytes is the record for leanCoP...
Needs occurs-check enabled!

```
p(M,I):-member(C,M),p([!],[[-!|C]|M],[],I).
p(C,M,P,I):-C=[];C=[L|G],(-N=L;-L=N)->(member(N,P);
\+length(P,I),member(D,M),copy_term(D,E),append(A,[N|B],E),
append(A,B,F),p(F,M,[L|P],I)),p(G,M,P,I).
```

Decision procedure for propositional logic and comparatively strong performance for first-order logic.

Prolog: leanTaP & leanCoP

leanTaP is written in Prolog and implements a theorem prover for first-order logic based on free-variable tableaux.

The unique thing about leanTaP is its small size.

The minimal version of the Prolog source code is only 360 bytes.

But recently 199 bytes is the record for leanCoP...
Needs occurs-check enabled!

```
p(M,I):-member(C,M),p([!],[[-!|C]|M],[],I).
p(C,M,P,I):-C=[];C=[L|G],(-N=L;-L=N)->(member(N,P);
\+length(P,I),member(D,M),copy_term(D,E),append(A,[N|B],E),
append(A,B,F),p(F,M,[L|P],I)),p(G,M,P,I).
```

Decision procedure for propositional logic and comparatively strong performance for first-order logic.

Source code available for popular Prolog systems, including SWI-Prolog, and easy to modify and/or integrate.

Gentzen System / Hilbert System

1.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), \neg q(a), \neg p(a), \exists x q(x), q(a)$	Axiom
2.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), p(a), \neg p(a), \exists x q(x), q(a)$	Axiom
3.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), \neg(p(a) \rightarrow q(a)), \neg p(a), \exists x q(x), q(a)$	$\beta \rightarrow$, 1, 2
4.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), \neg(p(a) \rightarrow q(a)), \neg p(a), \exists x q(x)$	γ, 3
5.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), \neg p(a), \exists x q(x)$	γ, 4
6.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), \neg\exists x p(x), \exists x q(x)$	δ, 5
7.	$\vdash \neg\forall x(p(x) \rightarrow q(x)), \exists x p(x) \rightarrow \exists x q(x)$	$\alpha \rightarrow$, 6
8.	$\vdash \forall x(p(x) \rightarrow q(x)) \rightarrow (\exists x p(x) \rightarrow \exists x q(x))$	$\alpha \rightarrow$, 7

1.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash \exists x p(x)$	Assumption
2.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash p(a)$	C-rule
3.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash \forall x(p(x) \rightarrow q(x))$	Assumption
4.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash p(a) \rightarrow q(a)$	Axiom 4
5.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash q(a)$	MP 2, 4
6.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash q(a) \rightarrow \exists x q(x)$	Theorem 8.14
7.	$\forall x(p(x) \rightarrow q(x)), \exists x p(x) \vdash \exists x q(x)$	MP 5, 6
8.	$\forall x(p(x) \rightarrow q(x)) \vdash \exists x p(x) \rightarrow \exists x q(x)$	Deduction
9.	$\vdash \forall x(p(x) \rightarrow q(x)) \rightarrow (\exists x p(x) \rightarrow \exists x q(x))$	Deduction

Skolemization

```
~(all(X, (p(X) => q(X))) => (ex(X, p(X)) => ex(X, q(X))))
```

```
~(Ax1(p(x1) => q(x1)) => (Ex1p(x1) => Ex1q(x1)))
~(Ax1(p(x1) => q(x1)) => (Ex2p(x2) => Ex3q(x3)))
~(~Ax1(~p(x1) \ q(x1)) \ (~Ex2p(x2) \ Ex3q(x3)))
(Ax1(~p(x1) \ q(x1)) & (Ex2p(x2) & Ax3~q(x3)))
Ax1Ex2Ax3((~p(x1) \ q(x1)) & (p(x2) & ~q(x3)))
Ax1Ex2Ax3((~p(x1) \ q(x1)) & (p(x2) & ~q(x3)))
Ax1Ax2((~p(x1) \ q(x1)) & (p(f(x1)) & ~q(x2)))
```

```
[~p(x),q(x)][p(f(x))][~q(x)]
```

The rename to `x` justified by the following theorem in the textbook:

$$\forall x(A(x) \wedge B(x)) \leftrightarrow (\forall x A(x) \wedge \forall x B(x))$$

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$ is a unifier.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$ is a unifier.

$\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$ is a simpler unifier.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$ is a unifier.

$\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$ is a simpler unifier.

$\{x \leftarrow f(a), z \leftarrow y\}$ is an mgu.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$ is a unifier.

$\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$ is a simpler unifier.

$\{x \leftarrow f(a), z \leftarrow y\}$ is an mgu.

$\{x \leftarrow f(a), y \leftarrow z\}$ is another mgu.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$ is a unifier.

$\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$ is a simpler unifier.

$\{x \leftarrow f(a), z \leftarrow y\}$ is an mgu.

$\{x \leftarrow f(a), y \leftarrow z\}$ is another mgu.

Unifiability of atoms is more conveniently described as a set of term equations.

Unification

Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical.

A *most general unifier* (mgu) is a unifier such that any other unifier can be obtained by a further substitution.

Consider the set of atoms $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$:

$\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$ is a unifier.

$\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$ is a simpler unifier.

$\{x \leftarrow f(a), z \leftarrow y\}$ is an mgu.

$\{x \leftarrow f(a), y \leftarrow z\}$ is another mgu.

Unifiability of atoms is more conveniently described as a set of term equations.

Use an alternative to Robinson's unification algorithm.

Example

```
?- test_unify(p(g(Y),f(X,h(X),Y)),p(X,f(g(Z),W,Z))).

Unify p(g(x1),f(x2,h(x2),x1))
 and  p(x2,f(g(x3),x4,x3))

x1 = x3
x2 = g(x3)
x4 = h(g(x3))

Yes
```

Martelli–Montanari Algorithm

Tries to find an mgu of a set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$.

Martelli–Montanari Algorithm

Tries to find an mgu of a set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$.

Nondeterministically choose from the set of equations an equation of a form below and perform the associated action.

(1)	$t = x$ where t is not a variable	*replace by the equation* $x = t$,
(2)	$x = x$	*delete the equation*,
(3')	$f(s_1, ..., s_n) = g(t_1, ..., t_m)$ where $f \neq g$	*halt with failure*,
(3)	$f(s_1, ..., s_n) = f(t_1, ..., t_n)$	*replace by the equations* $s_1 = t_1, ..., s_n = t_n$,
(4')	$x = t$ where x occurs in t and x differs from t	*halt with failure*,
(4)	$x = t$ where x does not occur in t and x occurs elsewhere	*apply the substitution* $\{x \leftarrow t\}$ *to all other equations.*

Martelli–Montanari Algorithm — Continued

The algorithm terminates when no action can be performed or when failure arises.

Martelli–Montanari Algorithm — Continued

The algorithm terminates when no action can be performed or when failure arises.

In case of success, by changing in the final set of equations all occurrences of "$=$" to "$\leftarrow$" the desired mgu is obtained.

Martelli–Montanari Algorithm — Continued

The algorithm terminates when no action can be performed or when failure arises.

In case of success, by changing in the final set of equations all occurrences of "$=$" to "$\leftarrow$" the desired mgu is obtained.

Note that action (3) includes the case $c = c$ for every constant c which leads to deletion of such an equation.

Martelli–Montanari Algorithm — Continued

The algorithm terminates when no action can be performed or when failure arises.

In case of success, by changing in the final set of equations all occurrences of "$=$" to "$\leftarrow$" the desired mgu is obtained.

Note that action (3) includes the case $c = c$ for every constant c which leads to deletion of such an equation.

In addition, action (3') includes the case of two different constants.

Martelli–Montanari Algorithm — Continued

The algorithm terminates when no action can be performed or when failure arises.

In case of success, by changing in the final set of equations all occurrences of "$=$" to "$\leftarrow$" the desired mgu is obtained.

Note that action (3) includes the case $c = c$ for every constant c which leads to deletion of such an equation.

In addition, action (3') includes the case of two different constants.

In the textbook each function f has a unique arity (and each predicate p has a unique arity too).

Propositional Resolution

Resolution rule: C_1, C_2 / $(C_1 - \{\ell\}) \cup (C_2 - \{\ell^c\})$
$(\ell \in C_1$, $\ell^c \in C_2)$

Propositional Resolution

Resolution rule: $C_1,\ C_2\ /\ (C_1 - \{\ell\}) \cup (C_2 - \{\ell^c\})$
$(\ell \in C_1,\ \ell^c \in C_2)$

The clauses C_1, C_2 are called *clashing clauses* (they clash on ℓ, ℓ^c) and are parent clauses of the child clause, the *resolvent clause*.

1.	p	
2.	$\overline{p}\,q$	
3.	$\overline{r}$	
4.	$\overline{p}\,\overline{q}\,r$	
5.	$\overline{p}\,\overline{q}$	3,4
6.	$\overline{p}$	5,2
7.	$\square$	6,1

Hence $(p \to q \to r) \to (p \to q) \to p \to r$ is proved.

Propositional Resolution

Resolution rule: $C_1,\ C_2\ /\ (C_1 - \{\ell\}) \cup (C_2 - \{\ell^c\})$
$(\ell \in C_1,\ \ell^c \in C_2)$

The clauses C_1, C_2 are called *clashing clauses* (they clash on ℓ, ℓ^c) and are parent clauses of the child clause, the *resolvent clause*.

1.	p	
2.	$\overline{p}\,q$	
3.	$\overline{r}$	
4.	$\overline{p}\,\overline{q}\,r$	
5.	$\overline{p}\,\overline{q}$	3,4
6.	$\overline{p}$	5,2
7.	$\Box$	6,1

Hence $(p \to q \to r) \to (p \to q) \to p \to r$ is proved.

Propositional resolution is sound and complete.

Example

```
?- qed(~ (all(X, p(f(X))) & all(X, ~ p(X)))).

~(Ax1p(f(x1)) & Ax1~p(x1))

Ax1Ax2(p(f(x1)) & ~p(x2))

[p(f(x1))][~p(x2)]

Resolve [p(f(x1))]
  and   [~p(x2)]

x2 = f(x1)

[][p(f(x1))][~p(x2)]

Yes
```

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

An mgu is $\{x' \leftarrow f(x)\}$ and $p(f(x))$ and $\neg p(f(x))$ resolve to $\square$.

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

An mgu is $\{x' \leftarrow f(x)\}$ and $p(f(x))$ and $\neg p(f(x))$ resolve to $\Box$.

Hence $\neg(\forall x p(f(x)) \wedge \forall x \neg p(x))$ is proved.

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

An mgu is $\{x' \leftarrow f(x)\}$ and $p(f(x))$ and $\neg p(f(x))$ resolve to $\Box$.

Hence $\neg(\forall x p(f(x)) \wedge \forall x \neg p(x))$ is proved.

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

An mgu is $\{x' \leftarrow f(x)\}$ and $p(f(x))$ and $\neg p(f(x))$ resolve to $\square$.

Hence $\neg(\forall x p(f(x)) \wedge \forall x \neg p(x))$ is proved.

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Resolution rule: C_1, C_2 / $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

An mgu is $\{x' \leftarrow f(x)\}$ and $p(f(x))$ and $\neg p(f(x))$ resolve to $\Box$.

Hence $\neg(\forall x p(f(x)) \wedge \forall x \neg p(x))$ is proved.

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Resolution rule: C_1, C_2 / $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

The clauses C_1, C_2 are called *clashing clauses* (they clash on L_1, L_2) and are parent clauses of the child clause, the *resolvent clause*.

General Resolution

Example: Resolve the two clauses $p(f(x))$ and $\neg p(x)$.

First rename variable of the second clause $\neg p(x')$.

An mgu is $\{x' \leftarrow f(x)\}$ and $p(f(x))$ and $\neg p(f(x))$ resolve to $\Box$.

Hence $\neg(\forall x p(f(x)) \wedge \forall x \neg p(x))$ is proved.

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Resolution rule: C_1, C_2 / $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

The clauses C_1, C_2 are called *clashing clauses* (they clash on L_1, L_2) and are parent clauses of the child clause, the *resolvent clause*.

General resolution is sound and complete.

Resolution Procedure

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Resolution Procedure

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Collapsing of identical literals in a clause is called factoring.

Resolution Procedure

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Collapsing of identical literals in a clause is called factoring.

Resolution rule: C_1, C_2 / $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

Resolution Procedure

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Collapsing of identical literals in a clause is called factoring.

Resolution rule: C_1, C_2 / $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

The clauses C_1, C_2 are called *clashing clauses* (they clash on L_1, L_2) and are parent clauses of the child clause, the *resolvent clause*.

Resolution Procedure

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Collapsing of identical literals in a clause is called factoring.

Resolution rule: $C_1,\ C_2\ /\ (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

The clauses C_1, C_2 are called *clashing clauses* (they clash on L_1, L_2) and are parent clauses of the child clause, the *resolvent clause*.

Start with a set of clause S_0 and assume S_i has been constructed.

Resolution Procedure

If $L = \{l_1, \dots, l_n\}$ then $L^c = \{l_1^c, \dots, l_n^c\}$.

Collapsing of identical literals in a clause is called factoring.

Resolution rule: $C_1,\ C_2\ /\ (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

The clauses C_1, C_2 are called *clashing clauses* (they clash on L_1, L_2) and are parent clauses of the child clause, the *resolvent clause*.

Start with a set of clause S_0 and assume S_i has been constructed.

Choose clashing clauses and terminate if resolvent clause $C = \square$.

$\boxed{S_0 \text{ is unsatisfiable}}$

Resolution Procedure

If $L = \{l_1, \ldots, l_n\}$ then $L^c = \{l_1^c, \ldots, l_n^c\}$.

Collapsing of identical literals in a clause is called factoring.

Resolution rule: $C_1,\ C_2 \ /\ (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$
(C_1, C_2 must have no variables in common and $L_1 \subseteq C_1$, $L_2 \subseteq C_2$ must be such that L_1 and L_2^c can be unified by an mgu σ)

The clauses C_1, C_2 are called *clashing clauses* (they clash on L_1, L_2) and are parent clauses of the child clause, the *resolvent clause*.

Start with a set of clause S_0 and assume S_i has been constructed.

Choose clashing clauses and terminate if resolvent clause $C = \Box$.

$$\boxed{S_0 \text{ is unsatisfiable}}$$

Otherwise construct $S_{i+1} = S_i \cup \{C\}$ and terminate if $S_{i+1} = S_i$ for all clashing clauses.

$$\boxed{S_0 \text{ is satisfiable}}$$

Resolution — Implementation

```
resolution(S) :- member([], S), !.
resolution(S) :-
  member(C1, S), member(C2, S), C1 \== C2,
  copy_term(C2, C2_R),
  clashing(C1, L1, C2_R, L2, Subst),
  delete_lit(C1, L1, Subst, C1P),
  delete_lit(C2_R, L2, Subst, C2P),
  clause_union(C1P, C2P, Resolvent),
  \+ clashing(Resolvent, _, Resolvent, _, _),
  \+ member(Resolvent, S),
  resolution([Resolvent | S]).
```

There is not factoring because it would complicate the code and anyway this naive implementation is quite likely to start searching infinite paths

Resolution — Remember

Factoring needed for completeness.

Resolution — Remember

Factoring needed for completeness.

Factoring can alternatively be seen as a separate rule.

Resolution — Remember

Factoring needed for completeness.

Factoring can alternatively be seen as a separate rule.

The resolution rule requires that the clauses have no variables in common.

Resolution — Remember

Factoring needed for completeness.

Factoring can alternatively be seen as a separate rule.

The resolution rule requires that the clauses have no variables in common.

Standardizing apart means that all the variables in one of the clauses are renamed before it is used in the resolution rule.

Resolution — Remember

Factoring needed for completeness.

Factoring can alternatively be seen as a separate rule.

The resolution rule requires that the clauses have no variables in common.

Standardizing apart means that all the variables in one of the clauses are renamed before it is used in the resolution rule.

Hyperresolution: Resolution on more than two clauses.