Automated Reasoning

Georg Moser

Institute of Computer Science @ UIBK

Winter 2013

Summary Last Lecture

Example

reachability is not expressible in first-order logic; that is, the class $\mathcal{K}_1$ of connected graphs is not Δ-elementary

Theorem

1. *compactness fails for second-order logic*
2. *Löwenheim-Skolem fails for second-order logic*
3. $\neg\,\exists$ *a calculus that is complete for second-order logic, in particular the set of valid second-order sentences is not recursively enumerable*

Example

$\exists$ set $\mathcal{H}$ of second-order sentences, such that $\mathrm{Mod}^{\mathrm{fin}}(\mathcal{H}) = \mathrm{NP}$

Outline of the Lecture

Early Approaches in Automated Reasoning

short recollection of Herbrand's theorem, Gilmore's prover, method of Davis and Putnam

Starting Points

resolution, tableau provers, structural Skolemisation, redundancy and deletion

Automated Reasoning with Equality

ordered resolution, paramodulation, ordered completion and proof orders, superposition

Applications of Automated Reasoning

Neuman-Stubblebinde Key Exchange Protocol, Robbins problem, resolution and paramodulation as decision procedure, . . .

Outline of the Lecture

Early Approaches in Automated Reasoning

short recollection of Herbrand's theorem, Gilmore's prover, method of Davis and Putnam

Starting Points

resolution, tableau provers, structural Skolemisation, redundancy and deletion

Automated Reasoning with Equality

ordered resolution, paramodulation, ordered completion and proof orders, superposition

Applications of Automated Reasoning

Neuman-Stubblebinde Key Exchange Protocol, Robbins problem, resolution and paramodulation as decision procedure, ...

Recall

Applications

1. Program Analysis
   logical products of interpretations allows the automated combination of simple interpreters

Recall

Applications

1. Program Analysis
   logical products of interpretations allows the automated combination of simple interpreters
2. Databases, in particular datalog
   datalog is a declarative language and syntactically it is a subset of Prolog; used in knowledge representation systems

Recall

Applications

1. Program Analysis
   logical products of interpretations allows the automated combination of simple interpreters
2. Databases, in particular datalog
   datalog is a declarative language and syntactically it is a subset of Prolog; used in knowledge representation systems
3. Types as Formulas
   the type checking in simple λ-calculus is equivalent to derivability in intuitionistic logic

Recall

Applications

1. Program Analysis
   logical products of interpretations allows the automated combination of simple interpreters
2. Databases, in particular datalog
   datalog is a declarative language and syntactically it is a subset of Prolog; used in knowledge representation systems
3. Types as Formulas
   the type checking in simple λ-calculus is equivalent to derivability in intuitionistic logic
4. Complexity Theory
   NP can be characterised as the class of existential second-order sentence

Additional Applications

Application ⑤: Issues of Security

- security protocols are small programs that aim at securing communications over a public network
- design of such protocols is difficult and error-prone
- we will study the use of a first-order theorem prover to show that the Neuman-Stubblebine key exchange protocol can be broken

Additional Applications

Application ⑤: Issues of Security

- security protocols are small programs that aim at securing communications over a public network
- design of such protocols is difficult and error-prone
- we will study the use of a first-order theorem prover to show that the Neuman-Stubblebine key exchange protocol can be broken

Application ⑥: Software Verification

- termination of programs is undecidable (Alan Turing)
- so what: termination of imperative programs can be shown by

  AProVE, Terminator, Julia, COSTA, . . .

  fully automatically . . .
- Terminator uses model-checking

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits
- in the last decade the approach was extended to the verification of properties of software

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits
- in the last decade the approach was extended to the verification of properties of software
- initially only safety properties could be analysed ("nothing bad happens")

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits
- in the last decade the approach was extended to the verification of properties of software
- initially only safety properties could be analysed ("nothing bad happens")
- recently liveness properties ("something good will happen") became of interest
- termination of a program is a liveness property

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits
- in the last decade the approach was extended to the verification of properties of software
- initially only safety properties could be analysed ("nothing bad happens")
- recently liveness properties ("something good will happen") became of interest
- termination of a program is a liveness property

Terminator research project

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits
- in the last decade the approach was extended to the verification of properties of software
- initially only safety properties could be analysed ("nothing bad happens")
- recently liveness properties ("something good will happen") became of interest
- termination of a program is a liveness property

Terminator research project

- developed by Microsoft Research Cambridge

Software Verification

- in the early years of model-checking mainly hardware was analysed like integrated circuits
- in the last decade the approach was extended to the verification of properties of software
- initially only safety properties could be analysed ("nothing bad happens")
- recently liveness properties ("something good will happen") became of interest
- termination of a program is a liveness property

Terminator research project

- developed by Microsoft Research Cambridge
- employs transition invariants, given a program step relation $\to_{\mathsf{P}}$ find finitely many well-founded relations $\mathsf{U}_1, \ldots, \mathsf{U}_n$ whose union contains the transitive closure of $\to_{\mathsf{P}}$

A Bit More on Java

Example

```
public static int div(int x, int y) {
  int res = 0;
  while (x >= y && y > 0) {
    x = x-y;
    res = res + 1;
  }
  return res;
}
```

A Bit More on Java

Example

```
public static int div(int x, int y) {
  int res = 0;
  while (x >= y && y > 0) {
    x = x-y;
    res = res + 1;
  }
  return res;
}
```

Termination of the example could be proven.

A Bit More on Java (cont'd)

Example

```
public static void test(int n, int m){
  if (0 < n && n < m) {
    int j = n+1;
    while(j<n || j > n){
      if (j>m) j=0 else j=j+1;
    }
  }
}
```

A Bit More on Java (cont'd)

Example

```
public static void test(int n, int m){
  if (0 < n && n < m) {
    int j = n+1;
    while(j<n || j > n){
      if (j>m) j=0 else j=j+1;
    }
  }
}
```

We were unable to show termination of the example.

Herbrand's Theorem

Jacques Herbrand (1908–1931) proposed to

- transform first-order into propositional logic
- basis of Gilmore's prover

$\mathcal{G}$ a set of universal sentences (of $\mathcal{L}$) without $=$

Theorem

$\mathcal{G}$ is satisfiable iff $\mathcal{G}$ has a Herbrand model (over $\mathcal{L}$)

Gilmore's Prover (declarative version)

1. F be an arbitrary sentence in language $\mathcal{L}$

Gilmore's Prover (declarative version)

1. F be an arbitrary sentence in language $\mathcal{L}$
2. consider its negation $\neg F$
   wlog $\neg F = \forall x_1 \cdots \forall x_n G(x_1, \dots, x_n)$ in SNF

Gilmore's Prover (declarative version)

1. F be an arbitrary sentence in language $\mathcal{L}$
2. consider its negation $\neg F$
   wlog $\neg F = \forall x_1 \cdots \forall x_n G(x_1, \ldots, x_n)$ in SNF
3. consider all possible Herbrand interpretations of $\mathcal{L}$

Gilmore's Prover (declarative version)

1. F be an arbitrary sentence in language $\mathcal{L}$
2. consider its negation $\neg F$
   wlog $\neg F = \forall x_1 \cdots \forall x_n G(x_1, \dots, x_n)$ in SNF
3. consider all possible Herbrand interpretations of $\mathcal{L}$
4. F is valid if $\exists$ finite unsatisfiable subset $S \subseteq \mathrm{Gr}(\neg F)$

Gilmore's Prover (declarative version)

1. F be an arbitrary sentence in language $\mathcal{L}$
2. consider its negation $\neg F$
   wlog $\neg F = \forall x_1 \cdots \forall x_n G(x_1, \ldots, x_n)$ in SNF
3. consider all possible Herbrand interpretations of $\mathcal{L}$
4. F is valid if $\exists$ finite unsatisfiable subset $S \subseteq \mathrm{Gr}(\neg F)$

$\mathcal{A} = \{A_0, A_1, A_2, \ldots\}$ be atomic formulas over Herbrand universe of $\mathcal{L}$

Gilmore's Prover (declarative version)

1. F be an arbitrary sentence in language $\mathcal{L}$
2. consider its negation $\neg F$
   wlog $\neg F = \forall x_1 \cdots \forall x_n G(x_1, \dots, x_n)$ in SNF
3. consider all possible Herbrand interpretations of $\mathcal{L}$
4. F is valid if $\exists$ finite unsatisfiable subset $S \subseteq \mathrm{Gr}(\neg F)$

$\mathcal{A} = \{A_0, A_1, A_2, \dots\}$ be atomic formulas over Herbrand universe of $\mathcal{L}$

Definition (Semantic Tree)

the semantic tree T for F:

- the root is a semantic tree
- let I be a node in T of height n; then I is either a
  1. leaf node or
  2. the edges e_1, e_2 leaving node I are labelled by A_n and $\neg A_n$

Fact

path in T gives rise to a (partial) Herbrand interpretation $\mathcal{I}$ of F'

Fact

path in T gives rise to a (partial) Herbrand interpretation $\mathcal{I}$ of F'

Definition

Fact

path in T gives rise to a (partial) Herbrand interpretation $\mathcal{I}$ of F'

Definition

- let $I \in T$, Herbrand interpretation induced by I is denoted as $\mathcal{I}$

Fact

path in T gives rise to a (partial) Herbrand interpretation $\mathcal{I}$ of F'

Definition

- let $I \in T$, Herbrand interpretation induced by I is denoted as $\mathcal{I}$
- I is closed, if $\exists\ G \in \mathrm{Gr}(\neg F)$ such that $\mathcal{I} \not\models G$ and thus $\mathcal{I} \not\models \neg F$

Fact

path in T gives rise to a (partial) Herbrand interpretation $\mathcal{I}$ of F'

Definition

- let $I \in T$, Herbrand interpretation induced by I is denoted as $\mathcal{I}$
- I is closed, if $\exists\ G \in \mathrm{Gr}(\neg F)$ such that $\mathcal{I} \not\models G$ and thus $\mathcal{I} \not\models \neg F$

Lemma

if all nodes in T are closed then F is valid

Fact

path in T gives rise to a (partial) Herbrand interpretation $\mathcal{I}$ of F'

Definition

- let $I \in T$, Herbrand interpretation induced by I is denoted as $\mathcal{I}$
- I is closed, if $\exists\ G \in \mathrm{Gr}(\neg F)$ such that $\mathcal{I} \not\models G$ and thus $\mathcal{I} \not\models \neg F$

Lemma

if all nodes in T are closed then F is valid

Proof.

- all nodes in T are closed
- $\exists$ finite unsatisfiable $S \subseteq \mathrm{Gr}(\neg F)$
- by Herbrand's theorem $\neg F$ is unsatisfiable, hence F is valid ∎

Gilmore's Prover

Definition

the Herbrand universe for a language $\mathcal{L}$ can be constructed iteratively as follows:

$$H_0 := \begin{cases} \{c \mid c \text{ is a constant in } \mathcal{L}\} & \exists \text{ constants} \\ \{c\} & \text{otherwise} \end{cases}$$

$$H_{n+1} := \{f(t_1, \dots, t_k) \mid f^k \in \mathcal{L}, t_1, \dots, t_k \in H_n\}$$

finally $H := \bigcup_{n \geqslant 0} H_n$ denotes the Herbrand universe for $\mathcal{L}$

Gilmore's Prover

Definition

the Herbrand universe for a language $\mathcal{L}$ can be constructed iteratively as follows:

$$H_0 := \begin{cases} \{c \mid c \text{ is a constant in } \mathcal{L}\} & \exists \text{ constants} \\ \{c\} & \text{otherwise} \end{cases}$$

$$H_{n+1} := \{f(t_1, \dots, t_k) \mid f^k \in \mathcal{L}, t_1, \dots, t_k \in H_n\}$$

finally $H := \bigcup_{n \geqslant 0} H_n$ denotes the Herbrand universe for $\mathcal{L}$

Definition

let $\mathcal{C}$ denote a set of clauses over $\mathcal{L}$; define $\mathcal{C}'_n$ as the ground instances of $\mathcal{C}$ using only terms from H_n[a]

[a] a clause is a disjunction of literals

Gilmore's Prover in Pseudo-Code

```
begin {
  contr := false;
  n := 0;
  while (not contr) do {
    D' := DNF(C'_n);
    contr := all constituents of D'
             contain complementary literals;
    n := n + 1;
   }
 }
```

Gilmore's Prover in Pseudo-Code

```
begin {
  contr := false;
  n := 0;
  while (not contr) do {
    D' := DNF(C'_n);
    contr := all constituents of D'
             contain complementary literals;
    n := n + 1;
   }
 }
```

Disadvantages

- generation of all $\mathcal{C}'_n$
- transformation to DNF
- did not yield actual proofs of simple (predicate logic) formulas

Definitions

- a clause C is called reduced, if every literal occurs at most once in C
- a clause set $\mathcal{C}$ is called reduced for tautologies, if every clause in $\mathcal{C}$ is reduced and does not contain complementary literals

Definitions

- a clause C is called reduced, if every literal occurs at most once in C
- a clause set $\mathcal{C}$ is called reduced for tautologies, if every clause in $\mathcal{C}$ is reduced and does not contain complementary literals

Definition (tautology rule)

delete all clauses containing complementary literals

Definitions

- a clause C is called reduced, if every literal occurs at most once in C
- a clause set $\mathcal{C}$ is called reduced for tautologies, if every clause in $\mathcal{C}$ is reduced and does not contain complementary literals

Definition (tautology rule)

delete all clauses containing complementary literals

let $\mathcal{C}'$ be ground and reduced for tautologies

Definitions

- a clause C is called reduced, if every literal occurs at most once in C
- a clause set $\mathcal{C}$ is called reduced for tautologies, if every clause in $\mathcal{C}$ is reduced and does not contain complementary literals

Definition (tautology rule)

delete all clauses containing complementary literals

let $\mathcal{C}'$ be ground and reduced for tautologies

Definition (one-literal rule)

let $C \in \mathcal{C}'$ and suppose

1. C consists of just one literal L
2. remove all clauses $D \in \mathcal{C}'$ such that L occurs in D
3. remove $\neg L$ from all remaining clauses in $\mathcal{C}'$

Definition (pure literal rule)

let $\mathcal{D}' \subseteq \mathcal{C}'$ such that

1. $\exists$ literal L that appears in all clauses in $\mathcal{D}'$
2. $\neg L$ doesn't appear in $\mathcal{C}'$
3. replace $\mathcal{C}'$ by $\mathcal{C}' \setminus \mathcal{D}'$

Definition (pure literal rule)

let $\mathcal{D}' \subseteq \mathcal{C}'$ such that

1. $\exists$ literal L that appears in all clauses in $\mathcal{D}'$
2. $\neg L$ doesn't appear in $\mathcal{C}'$
3. replace $\mathcal{C}'$ by $\mathcal{C}' \setminus \mathcal{D}'$

Definition (splitting rule)

suppose the clause set $\mathcal{C}'$ can be written as
$\mathcal{C}' = \{A_1, \dots, A_n, B_1, \dots, B_m\} \cup \mathcal{D}$ where

1. $\exists$ literal L, such that neither L nor $\neg L$ occurs in $\mathcal{D}$
2. L occurs in any A_i (but in no B_j); A_i' is the result of removing L
3. $\neg L$ occurs in any B_j (but in no A_i) B_j' is the result of removing $\neg L$
4. rule consists in splitting $\mathcal{C}'$ into $\mathcal{C}_1' := \{A_1', \dots, A_n'\} \cup \mathcal{D}$ and $\mathcal{C}_2' := \{B_1', \dots, B_m'\} \cup \mathcal{D}$

The Method of Davis and Putnam

Definition (DPLL Method)

the method encompasses the above defined four rules

- tautology rule
- one-literal rule
- pure literal rule
- splitting rule

The Method of Davis and Putnam

Definition (DPLL Method)

the method encompasses the above defined four rules

- tautology rule
- one-literal rule
- pure literal rule
- splitting rule

Theorem

1. *the rules of the DPLL-method are correct*
2. *that is, if $\mathcal{D}$ is a set of ground clauses and either $\mathcal{D}'$ or $\mathcal{D}_1$ and $\mathcal{D}_2$ are obtained by the above rules, then $\mathcal{D}$ is satisfiable if $\mathcal{D}'$ ($\mathcal{D}_1$ or $\mathcal{D}_2$) is satisfiable*

DPLL-tree and DPLL-decision tree

let $\mathcal{C}'$ be a set of reduced ground clauses

Definition

- T consists only of the root, labelled by $\mathcal{C}'$
- let N be a node in T, labelled by $\mathcal{D}$; then N is either a
  1. leaf node,
  2. N has one successor N', labelled by $\mathcal{D}'$, where $\mathcal{D}'$ is obtained as the application of tautology, one-literal, pure literal rule to $\mathcal{D}$, or
  3. N has two successors N_1, N_2 labelled by the clause sets obtained by an application of the split rule to $\mathcal{D}$

DPLL-tree and DPLL-decision tree

let $\mathcal{C}'$ be a set of reduced ground clauses

Definition

- T consists only of the root, labelled by $\mathcal{C}'$
- let N be a node in T, labelled by $\mathcal{D}$; then N is either a
  1. leaf node,
  2. N has one successor N', labelled by $\mathcal{D}'$, where $\mathcal{D}'$ is obtained as the application of tautology, one-literal, pure literal rule to $\mathcal{D}$, or
  3. N has two successors N_1, N_2 labelled by the clause sets obtained by an application of the split rule to $\mathcal{D}$

Definition (DPLL-decision tree)

a DPLL-tree is a decision tree for $\mathcal{C}'$ if

1. all leafs are labelled by the empty clause $\Box$, or
2. $\exists$ leaf labelled by the empty clause set $\varnothing$

Theorem

- *let* $\mathcal{C}'$ *be a reduced set of ground clauses and let* T *be a decision tree proving satisfiability or unsatisfiability for* $\mathcal{C}'$
- *then* $\mathcal{C}'$ *is satisfiable or unsatisfiable, respectively*

Theorem

- *let $\mathcal{C}'$ be a reduced set of ground clauses and let T be a decision tree proving satisfiability or unsatisfiability for $\mathcal{C}'$*
- *then $\mathcal{C}'$ is satisfiable or unsatisfiable, respectively*

Theorem

- *let $\mathcal{C}'$ be as above and let T be a DPLL-tree for $\mathcal{C}'$*
- *then T can be extended to a decision tree for $\mathcal{C}'$*

Theorem

- *let $\mathcal{C}'$ be a reduced set of ground clauses and let T be a decision tree proving satisfiability or unsatisfiability for $\mathcal{C}'$*
- *then $\mathcal{C}'$ is satisfiable or unsatisfiable, respectively*

Theorem

- *let $\mathcal{C}'$ be as above and let T be a DPLL-tree for $\mathcal{C}'$*
- *then T can be extended to a decision tree for $\mathcal{C}'$*

Proof

by induction on the number ℓ of atoms in $\mathcal{C}'$

Theorem

- *let $\mathcal{C}'$ be a reduced set of ground clauses and let T be a decision tree proving satisfiability or unsatisfiability for $\mathcal{C}'$*
- *then $\mathcal{C}'$ is satisfiable or unsatisfiable, respectively*

Theorem

- *let $\mathcal{C}'$ be as above and let T be a DPLL-tree for $\mathcal{C}'$*
- *then T can be extended to a decision tree for $\mathcal{C}'$*

Proof

by induction on the number ℓ of atoms in $\mathcal{C}'$

1. $\ell = 0$: $\mathcal{C}'$ is either empty or contains $\square$, T is already a decision tree

Theorem

- *let $\mathcal{C}'$ be a reduced set of ground clauses and let T be a decision tree proving satisfiability or unsatisfiability for $\mathcal{C}'$*
- *then $\mathcal{C}'$ is satisfiable or unsatisfiable, respectively*

Theorem

- *let $\mathcal{C}'$ be as above and let T be a DPLL-tree for $\mathcal{C}'$*
- *then T can be extended to a decision tree for $\mathcal{C}'$*

Proof

by induction on the number ℓ of atoms in $\mathcal{C}'$

1. $\ell = 0$: $\mathcal{C}'$ is either empty or contains $\Box$, T is already a decision tree
2. $\ell > 0$: we distinguish
   - T consists only of the root, labelled by $\mathcal{C}'$
   - T contains more than one node

Proof (cont'd).

- T consists only of the root, labelled by $\mathcal{C}'$

Proof (cont'd).

- T consists only of the root, labelled by $\mathcal{C}'$
  we employ a one-literal, pure literal rule, or a splitting rule; extend T such that the successors nodes are labelled with smaller clause sets; induction hypothesis becomes applicable

Proof (cont'd).

- T consists only of the root, labelled by $\mathcal{C}'$
  we employ a one-literal, pure literal rule, or a splitting rule; extend T such that the successors nodes are labelled with smaller clause sets; induction hypothesis becomes applicable
- T contains more than one node

Proof (cont'd).

- T consists only of the root, labelled by $\mathcal{C}'$
  we employ a one-literal, pure literal rule, or a splitting rule; extend T such that the successors nodes are labelled with smaller clause sets; induction hypothesis becomes applicable
- T contains more than one node
  let $\mathcal{D}_1, \dots, \mathcal{D}_n$ denote all leaf nodes of T; for at least one of these nodes we can emply one-literal, pure literal rule, or a splitting rule; then we argue as in the first sub-case

Proof (cont'd).

- T consists only of the root, labelled by $\mathcal{C}'$
  we employ a one-literal, pure literal rule, or a splitting rule; extend T such that the successors nodes are labelled with smaller clause sets; induction hypothesis becomes applicable
- T contains more than one node
  let $\mathcal{D}_1, \ldots, \mathcal{D}_n$ denote all leaf nodes of T; for at least one of these nodes we can emply one-literal, pure literal rule, or a splitting rule; then we argue as in the first sub-case

∎

Proof (cont'd).

- T consists only of the root, labelled by $\mathcal{C}'$
  we employ a one-literal, pure literal rule, or a splitting rule; extend T such that the successors nodes are labelled with smaller clause sets; induction hypothesis becomes applicable
- T contains more than one node
  let $\mathcal{D}_1, \dots, \mathcal{D}_n$ denote all leaf nodes of T; for at least one of these nodes we can emply one-literal, pure literal rule, or a splitting rule; then we argue as in the first sub-case ∎

Definition

DPLL(a) remove multiple occurrences of literals in $\mathcal{C}'$ to obtain a reduced clause set $\mathcal{D}_1$

DPLL(b) apply the tautology rule exhaustively to $\mathcal{D}_1$ to obtain a reduced clause set $\mathcal{D}_2$ that is reduced for tautologies

Definition

DPLL(c) construct a decision tree for $\mathcal{D}_2$.

Definition

DPLL(c) construct a decision tree for $\mathcal{D}_2$.

Method of Davis and Putnam in Pseudo-Code

```
if C does not contain function symbols
then apply DPLL(a)-DPLL(c) on C'_0
else {
  n := 0;
  contr := false;
  while (¬ contr) do {
    apply DPLL(a)-DPLL(c) on C'_n;
    if the decision tree proves unsatisfiability,
    then contr := true
    else contr := false;
    n := n + 1;
  }}
```

The Language of Clause Logic (with Equality)

Definition

- individual constants
  $k_0, k_1, \dots, k_j, \dots$ denoted c, d, etc.
- function constants with i arguments
  $f^i_0, f^i_1, \dots, f^i_j, \dots$ denoted f, g, h, etc.
- predicate constants with i arguments
  $R^i_0, R^i_1, \dots, R^i_j, \dots$ denoted P, Q, R, etc.
- variables, collected in $\mathcal{V}$
  $x_0, x_1, \dots, x_j, \dots$ denoted x, y, z, etc.

The Language of Clause Logic (with Equality)

Definition

- individual constants
  $k_0, k_1, \dots, k_j, \dots$ denoted c, d, etc.
- function constants with i arguments
  $f^i_0, f^i_1, \dots, f^i_j, \dots$ denoted f, g, h, etc.
- predicate constants with i arguments
  $R^i_0, R^i_1, \dots, R^i_j, \dots$ denoted P, Q, R, etc.
- variables, collected in $\mathcal{V}$
  $x_0, x_1, \dots, x_j, \dots$ denoted x, y, z, etc.

Definition

- propositional connectives $\neg$, $\vee$
- equality sign $=$

Definition

1. $P(t_1, \ldots, t_n)$ is called an atomic formula if $t_1, \ldots, t_n$ are terms, P a predicate constant

Definition

1. $P(t_1, \ldots, t_n)$ is called an atomic formula if $t_1, \ldots, t_n$ are terms, P a predicate constant
2. a literal is an atomic formula or its negation

Definition

1. $P(t_1, \dots, t_n)$ is called an atomic formula if $t_1, \dots, t_n$ are terms, P a predicate constant
2. a literal is an atomic formula or its negation
3. a clause is disjunction of literals

Definition

1. $P(t_1, \ldots, t_n)$ is called an atomic formula if $t_1, \ldots, t_n$ are terms, P a predicate constant
2. a literal is an atomic formula or its negation
3. a clause is disjunction of literals

Theorem

$\forall$ *first-order sentence* F, $\exists$ *set of clauses* $\mathcal{C} = \{C_1, \ldots, C_m\}$

$$F \approx \forall x_1 \cdots \forall x_n (C_1 \wedge \cdots \wedge C_m)$$

Definition

1. $P(t_1, \ldots, t_n)$ is called an atomic formula if $t_1, \ldots, t_n$ are terms, P a predicate constant
2. a literal is an atomic formula or its negation
3. a clause is disjunction of literals

Theorem

$\forall$ first-order sentence F, $\exists$ set of clauses $\mathcal{C} = \{C_1, \ldots, C_m\}$

$$F \approx \forall x_1 \cdots \forall x_n (C_1 \wedge \cdots \wedge C_m)$$

Proof.

- let F be a sentence (in standard first-order language)
- there exists $G \approx F$ such that

$$G = \forall x_1 \cdots \forall x_n (H_1(x_1, \ldots, x_n) \wedge \cdots \wedge H_m(x_1, \ldots, x_n))$$

- each H_i $(i = 1, \ldots, m)$ is a disjunction of literals, hence a clause ∎

Definition

1. $\square$ is a clause
2. literals are clauses
3. if C, D are clauses, then $C \vee D$ is a clause

Definition

1. $\square$ is a clause
2. literals are clauses
3. if C, D are clauses, then $C \vee D$ is a clause

Convention

we use (i) the equivalences $A \equiv \neg\neg A$, A atomic formula, that (ii) disjunction $\vee$ is associative and commutative, and (iii) $\square \vee \square = \square$, and $C \vee \square = \square \vee C = C$

Definition

1. $\square$ is a clause
2. literals are clauses
3. if C, D are clauses, then $C \vee D$ is a clause

Convention

we use (i) the equivalences $A \equiv \neg\neg A$, A atomic formula, that (ii) disjunction $\vee$ is associative and commutative, and (iii) $\square \vee \square = \square$, and $C \vee \square = \square \vee C = C$

Definition

- let $\mathcal{T}$ denote the set of terms in our language
- $\mathcal{V}\mathrm{ar}(E)$ denotes set of variables occurring in E
- a substitution σ is a mapping $\mathcal{V} \to \mathcal{T}$ such that $\sigma(x) = x$, for almost all x
- we write $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$; empty subst. denoted by ϵ

Most General Unifier

application of a substitution σ to expression E is denoted as $E\sigma$; $E\sigma$ is called an instance of E

Definition

- $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, $\tau = \{y_1 \mapsto r_1, \ldots, y_1 \mapsto r_m\}$
- composition of σ and τ denoted as $\sigma\tau$:

  $$\{x_1 \mapsto t_1\tau, \ldots, x_n \mapsto t_n\tau\} \cup \{y_i \mapsto r_i \mid \text{for all } j = 1, \ldots, n,\ y_i \neq x_j\}$$

- σ is more general than a substitution τ, if there exists a substitution ρ such that $\sigma\rho = \tau$

Most General Unifier

application of a substitution σ to expression E is denoted as $E\sigma$; $E\sigma$ is called an instance of E

Definition

- $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, $\tau = \{y_1 \mapsto r_1, \ldots, y_1 \mapsto r_m\}$
- composition of σ and τ denoted as $\sigma\tau$:

  $$\{x_1 \mapsto t_1\tau, \ldots, x_n \mapsto t_n\tau\} \cup \{y_i \mapsto r_i \mid \text{for all } j = 1, \ldots, n,\ y_i \neq x_j\}$$

- σ is more general than a substitution τ, if there exists a substitution ρ such that $\sigma\rho = \tau$ $\quad$ $E\tau$ is instance of $E\sigma$

Most General Unifier

application of a substitution σ to expression E is denoted as $E\sigma$; $E\sigma$ is called an instance of E

Definition

- $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, $\tau = \{y_1 \mapsto r_1, \dots, y_1 \mapsto r_m\}$
- composition of σ and τ denoted as $\sigma\tau$:

$$\{x_1 \mapsto t_1\tau, \dots, x_n \mapsto t_n\tau\} \cup \{y_i \mapsto r_i \mid \text{for all } j = 1, \dots, n,\ y_i \neq x_j\}$$

- σ is more general than a substitution τ, if there exists a substitution ρ such that $\sigma\rho = \tau$ $E\tau$ is instance of $E\sigma$

Definition

- a substitution σ such that $E\sigma = F\sigma$ is unifier of E, F
  generalises to sets U of expressions ($=$ terms or atomic formulas)
- unifier σ is most general if σ is more general than any other unifier

Example

consider $U = \{\mathsf{P}(x, \mathsf{f}(x)), \mathsf{P}(y, \mathsf{f}(x)), \mathsf{P}(x', y')\}$

- $\sigma = \{x \mapsto 0, y \mapsto 0, x' \mapsto 0, y' \mapsto \mathsf{f}(0)\}$ is a unifier of U
- $\tau = \{y \mapsto x, x' \mapsto x, y' \mapsto \mathsf{f}(x)\}$ is most general

Example

consider $U = \{\mathsf{P}(x, \mathsf{f}(x)), \mathsf{P}(y, \mathsf{f}(x)), \mathsf{P}(x', y')\}$

- $\sigma = \{x \mapsto 0, y \mapsto 0, x' \mapsto 0, y' \mapsto \mathsf{f}(0)\}$ is a unifier of U
- $\tau = \{y \mapsto x, x' \mapsto x, y' \mapsto \mathsf{f}(x)\}$ is most general

Definition

- sequence $E = u_1 \stackrel{?}{=} v_1, \dots, u_n \stackrel{?}{=} v_n$ is called an equality problem

Example

consider $U = \{\mathsf{P}(x, \mathsf{f}(x)), \mathsf{P}(y, \mathsf{f}(x)), \mathsf{P}(x', y')\}$

- $\sigma = \{x \mapsto 0, y \mapsto 0, x' \mapsto 0, y' \mapsto \mathsf{f}(0)\}$ is a unifier of U
- $\tau = \{y \mapsto x, x' \mapsto x, y' \mapsto \mathsf{f}(x)\}$ is most general

Definition

- sequence $E = u_1 \stackrel{?}{=} v_1, \dots, u_n \stackrel{?}{=} v_n$ is called an equality problem
- unifier of E is the unifier of $\{u_1 = v_1, \dots, u_n = v_n\}$

Example

consider $U = \{\mathsf{P}(x, \mathsf{f}(x)), \mathsf{P}(y, \mathsf{f}(x)), \mathsf{P}(x', y')\}$

- $\sigma = \{x \mapsto 0, y \mapsto 0, x' \mapsto 0, y' \mapsto \mathsf{f}(0)\}$ is a unifier of U
- $\tau = \{y \mapsto x, x' \mapsto x, y' \mapsto \mathsf{f}(x)\}$ is most general

Definition

- sequence $E = u_1 \stackrel{?}{=} v_1, \dots, u_n \stackrel{?}{=} v_n$ is called an equality problem
- unifier of E is the unifier of $\{u_1 = v_1, \dots, u_n = v_n\}$
- If $E = x_1 \stackrel{?}{=} v_1, \dots, x_n \stackrel{?}{=} v_n$, with x_i pairwise distinct and $x_i \notin \mathcal{V}\mathrm{ar}(v_j)$, then E is in solved form

Example

consider $U = \{\mathsf{P}(x, \mathsf{f}(x)), \mathsf{P}(y, \mathsf{f}(x)), \mathsf{P}(x', y')\}$

- $\sigma = \{x \mapsto 0, y \mapsto 0, x' \mapsto 0, y' \mapsto \mathsf{f}(0)\}$ is a unifier of U
- $\tau = \{y \mapsto x, x' \mapsto x, y' \mapsto \mathsf{f}(x)\}$ is most general

Definition

- sequence $E = u_1 \stackrel{?}{=} v_1, \dots, u_n \stackrel{?}{=} v_n$ is called an equality problem
- unifier of E is the unifier of $\{u_1 = v_1, \dots, u_n = v_n\}$
- If $E = x_1 \stackrel{?}{=} v_1, \dots, x_n \stackrel{?}{=} v_n$, with x_i pairwise distinct and $x_i \notin \mathcal{V}\mathsf{ar}(v_j)$, then E is in solved form

Example

U becomes $\quad \mathsf{P}(x, \mathsf{f}(x)) \stackrel{?}{=} \mathsf{P}(y, \mathsf{f}(x)), \mathsf{P}(y, \mathsf{f}(x)) \stackrel{?}{=} \mathsf{P}(x', y')$

τ becomes $\quad y \stackrel{?}{=} x, x' \stackrel{?}{=} x, y' \stackrel{?}{=} \mathsf{f}(x)$

Unification Algorithm

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \perp \qquad f \neq g$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \perp \qquad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \qquad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \perp \qquad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \qquad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

$$x \stackrel{?}{=} v, E \Rightarrow \perp \qquad x \neq v, x \in \mathcal{V}\mathrm{ar}(v)$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \perp \qquad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \qquad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

$$x \stackrel{?}{=} v, E \Rightarrow \perp \qquad x \neq v, x \in \mathcal{V}\mathrm{ar}(v)$$

$$v \stackrel{?}{=} x, E \Rightarrow x \stackrel{?}{=} v, E \qquad v \notin \mathcal{V}$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \dots, s_n) \stackrel{?}{=} f(t_1, \dots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \dots, s_n) \stackrel{?}{=} g(t_1, \dots, t_n), E \Rightarrow \perp \quad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \quad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

$$x \stackrel{?}{=} v, E \Rightarrow \perp \quad x \neq v, x \in \mathcal{V}\mathrm{ar}(v)$$

$$v \stackrel{?}{=} x, E \Rightarrow x \stackrel{?}{=} v, E \quad v \notin \mathcal{V}$$

Example

$$\mathsf{f}(x, \mathsf{g}(y), x) \stackrel{?}{=} \mathsf{f}(z, \mathsf{g}(x'), \mathsf{h}(x')) \Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), x \stackrel{?}{=} \mathsf{h}(x')$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \bot \qquad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \qquad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

$$x \stackrel{?}{=} v, E \Rightarrow \bot \qquad x \neq v, x \in \mathcal{V}\mathrm{ar}(v)$$

$$v \stackrel{?}{=} x, E \Rightarrow x \stackrel{?}{=} v, E \qquad v \notin \mathcal{V}$$

Example

$$\begin{aligned}\mathsf{f}(x, \mathsf{g}(y), x) \stackrel{?}{=} \mathsf{f}(z, \mathsf{g}(x'), \mathsf{h}(x')) &\Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), x \stackrel{?}{=} \mathsf{h}(x') \\ &\Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), z \stackrel{?}{=} \mathsf{h}(x')\end{aligned}$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \perp \quad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \quad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

$$x \stackrel{?}{=} v, E \Rightarrow \perp \quad x \neq v, x \in \mathcal{V}\mathrm{ar}(v)$$

$$v \stackrel{?}{=} x, E \Rightarrow x \stackrel{?}{=} v, E \quad v \notin \mathcal{V}$$

Example

$$\begin{aligned} \mathsf{f}(x, \mathsf{g}(y), x) \stackrel{?}{=} \mathsf{f}(z, \mathsf{g}(x'), \mathsf{h}(x')) &\Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), x \stackrel{?}{=} \mathsf{h}(x') \\ &\Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), z \stackrel{?}{=} \mathsf{h}(x') \\ &\Rightarrow x \stackrel{?}{=} z, y \stackrel{?}{=} x', z \stackrel{?}{=} \mathsf{h}(x') \end{aligned}$$

Unification Algorithm

$$u \stackrel{?}{=} u, E \Rightarrow E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n), E \Rightarrow s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n, E$$

$$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_n), E \Rightarrow \perp \quad f \neq g$$

$$x \stackrel{?}{=} v, E \Rightarrow x \stackrel{?}{=} v, E\{x \mapsto v\} \quad x \in \mathcal{V}\mathrm{ar}(E), x \notin \mathcal{V}\mathrm{ar}(v)$$

$$x \stackrel{?}{=} v, E \Rightarrow \perp \quad x \neq v, x \in \mathcal{V}\mathrm{ar}(v)$$

$$v \stackrel{?}{=} x, E \Rightarrow x \stackrel{?}{=} v, E \quad v \notin \mathcal{V}$$

Example

$$\begin{aligned}
\mathsf{f}(x, \mathsf{g}(y), x) \stackrel{?}{=} \mathsf{f}(z, \mathsf{g}(x'), \mathsf{h}(x')) &\Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), x \stackrel{?}{=} \mathsf{h}(x') \\
&\Rightarrow x \stackrel{?}{=} z, \mathsf{g}(y) \stackrel{?}{=} \mathsf{g}(x'), z \stackrel{?}{=} \mathsf{h}(x') \\
&\Rightarrow x \stackrel{?}{=} z, y \stackrel{?}{=} x', z \stackrel{?}{=} \mathsf{h}(x') \\
&\Rightarrow x \stackrel{?}{=} \mathsf{h}(x'), y \stackrel{?}{=} x', z \stackrel{?}{=} \mathsf{h}(x')
\end{aligned}$$

Definition

let $E = x_1 \stackrel{?}{=} v_1, \ldots, x_n \stackrel{?}{=} v_n$ be a equality problem in solved form
E induces substitution $\sigma_E = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$

Definition

let $E = x_1 \stackrel{?}{=} v_1, \ldots, x_n \stackrel{?}{=} v_n$ be a equality problem in solved form
E induces substitution $\sigma_E = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$

Theorem

1. *equality problems E is unifiable iff the unification algorithm stops with a solved form*
2. *if $E \Rightarrow^* E'$ such that E' is a solved form, then $\sigma_{E'}$ is a most general unifier (mgu for short) of E;*

Definition

let $E = x_1 \stackrel{?}{=} v_1, \dots, x_n \stackrel{?}{=} v_n$ be a equality problem in solved form
E induces substitution $\sigma_E = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$

Theorem

1. *equality problems E is unifiable iff the unification algorithm stops with a solved form*
2. *if $E \Rightarrow^* E'$ such that E' is a solved form, then $\sigma_{E'}$ is a most general unifier (mgu for short) of E;*

Proof.

in proof, we verify the following three facts:

Definition

let $E = x_1 \stackrel{?}{=} v_1, \dots, x_n \stackrel{?}{=} v_n$ be a equality problem in solved form
E induces substitution $\sigma_E = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$

Theorem

1. *equality problems E is unifiable iff the unification algorithm stops with a solved form*
2. *if $E \Rightarrow^* E'$ such that E' is a solved form, then $\sigma_{E'}$ is a most general unifier (mgu for short) of E;*

Proof.

in proof, we verify the following three facts:

- if $E \Rightarrow E'$, then σ is a unifier of E iff σ is a unifier of E'

Definition

let $E = x_1 \stackrel{?}{=} v_1, \dots, x_n \stackrel{?}{=} v_n$ be a equality problem in solved form
E induces substitution $\sigma_E = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$

Theorem

1. *equality problems E is unifiable iff the unification algorithm stops with a solved form*
2. *if $E \Rightarrow^* E'$ such that E' is a solved form, then $\sigma_{E'}$ is a most general unifier (mgu for short) of E;*

Proof.

in proof, we verify the following three facts:

- if $E \Rightarrow E'$, then σ is a unifier of E iff σ is a unifier of E'
- if $E \Rightarrow^* \bot$, then E is not unifiable

Definition

let $E = x_1 \stackrel{?}{=} v_1, \dots, x_n \stackrel{?}{=} v_n$ be a equality problem in solved form
E induces substitution $\sigma_E = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$

Theorem

1. *equality problems E is unifiable iff the unification algorithm stops with a solved form*
2. *if $E \Rightarrow^* E'$ such that E' is a solved form, then $\sigma_{E'}$ is a most general unifier (mgu for short) of E;*

Proof.

in proof, we verify the following three facts:

- if $E \Rightarrow E'$, then σ is a unifier of E iff σ is a unifier of E'
- if $E \Rightarrow^* \perp$, then E is not unifiable
- if $E \Rightarrow^* E'$ such that E' is a solved form, then $\sigma_{E'}$ is a mgu of E ∎

Resolution Calculus for First-Order Logic

Definition

resolution

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

factoring

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

σ is a mgu of the atomic formulas A and B

Resolution Calculus for First-Order Logic

restricted to atoms

Definition

resolution

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

factoring

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

σ is a mgu of the atomic formulas A and B

Resolution Calculus for First-Order Logic

Definition

resolution

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

factoring

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

σ is a mgu of the atomic formulas A and B

let $\mathcal{C}$ be a set of clauses; define resolution operator $\mathrm{Res}(\mathcal{C})$

Resolution Calculus for First-Order Logic

Definition

resolution

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

factoring

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

σ is a mgu of the atomic formulas A and B

let $\mathcal{C}$ be a set of clauses; define resolution operator $\mathrm{Res}(\mathcal{C})$

- $\mathrm{Res}(\mathcal{C}) = \{D \mid D \text{ is resolvent or factor with premises in } \mathcal{C}\}$
- $\mathrm{Res}^0(\mathcal{C}) = \mathcal{C}$; $\mathrm{Res}^{n+1}(\mathcal{C}) := \mathrm{Res}^n(\mathcal{C}) \cup \mathrm{Res}(\mathrm{Res}^n(\mathcal{C}))$
- $\mathrm{Res}^*(\mathcal{C}) := \bigcup_{n \geqslant 0} \mathrm{Res}^n(\mathcal{C})$

Resolution Calculus for First-Order Logic

Definition

resolution

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

factoring

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

σ is a mgu of the atomic formulas A and B

let $\mathcal{C}$ be a set of clauses; define resolution operator $\mathrm{Res}(\mathcal{C})$

- $\mathrm{Res}(\mathcal{C}) = \{D \mid D \text{ is resolvent or factor with premises in } \mathcal{C}\}$
- $\mathrm{Res}^0(\mathcal{C}) = \mathcal{C}$; $\mathrm{Res}^{n+1}(\mathcal{C}) := \mathrm{Res}^n(\mathcal{C}) \cup \mathrm{Res}(\mathrm{Res}^n(\mathcal{C}))$
- $\mathrm{Res}^*(\mathcal{C}) := \bigcup_{n \geqslant 0} \mathrm{Res}^n(\mathcal{C})$

Example

$$\frac{\mathsf{P}(\mathsf{x}) \vee \mathsf{Q}(\mathsf{f}(x, \mathsf{g}(y), x)) \quad \mathsf{R}(\mathsf{a}, \mathsf{b}) \vee \neg\mathsf{Q}(\mathsf{f}(z, \mathsf{g}(x'), \mathsf{h}(x')))}{\mathsf{P}(\mathsf{h}(x')) \vee \mathsf{R}(\mathsf{a}, \mathsf{b})} \; \{x \mapsto \mathsf{h}(x')\}$$

Soundness of Resolution

Definition

- if $\mathrm{Res}(\mathcal{C}) \subseteq \mathcal{C}$, then the clause set $\mathcal{C}$ is called saturated

Soundness of Resolution

Definition

- if $\mathrm{Res}(\mathcal{C}) \subseteq \mathcal{C}$, then the clause set $\mathcal{C}$ is called saturated
- let $\Box \notin \mathrm{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Soundness of Resolution

Definition

- if $\mathrm{Res}(\mathcal{C}) \subseteq \mathcal{C}$, then the clause set $\mathcal{C}$ is called saturated
- let $\square \notin \mathrm{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Theorem

resolution is sound: if F a sentence and $\mathcal{C}$ its clause form such that $\square \in \mathrm{Res}^*(\mathcal{C})$, then F is unsatisfiable

Soundness of Resolution

Definition

- if $\mathrm{Res}(\mathcal{C}) \subseteq \mathcal{C}$, then the clause set $\mathcal{C}$ is called saturated
- let $\square \notin \mathrm{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Theorem

resolution is sound: if F a sentence and $\mathcal{C}$ its clause form such that $\square \in \mathrm{Res}^*(\mathcal{C})$, then F is unsatisfiable

Proof.

- the theorem follows by case-distinction on the inferences

Soundness of Resolution

Definition

- if $\text{Res}(\mathcal{C}) \subseteq \mathcal{C}$, then the clause set $\mathcal{C}$ is called saturated
- let $\square \notin \text{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Theorem

resolution is sound: if F a sentence and $\mathcal{C}$ its clause form such that $\square \in \text{Res}^*(\mathcal{C})$, then F is unsatisfiable

Proof.

- the theorem follows by case-distinction on the inferences
- for each inference one verifies that if the assumptions (as formulas) are modelled by an interpretation $\mathcal{M}$, then the consequence holds in $\mathcal{M}$ as well

Soundness of Resolution

Definition

- if $\text{Res}(\mathcal{C}) \subseteq \mathcal{C}$, then the clause set $\mathcal{C}$ is called saturated
- let $\square \notin \text{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Theorem

resolution is sound: if F a sentence and $\mathcal{C}$ its clause form such that $\square \in \text{Res}^*(\mathcal{C})$, then F is unsatisfiable

Proof.

- the theorem follows by case-distinction on the inferences
- for each inference one verifies that if the assumptions (as formulas) are modelled by an interpretation $\mathcal{M}$, then the consequence holds in $\mathcal{M}$ as well

∎

Completeness of Resolution

Definitions

- a clause is called ground if it doesn't contain variables
- a ground substitution is a substitution whose range contains only terms without variables
- let $\square \notin \mathrm{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Completeness of Resolution

Definitions

- a clause is called ground if it doesn't contain variables
- a ground substitution is a substitution whose range contains only terms without variables
- let $\square \notin \mathrm{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Lemma

- *let S denote the set of all consistent ground clause sets*
- *then S has the satisfaction properties*

Completeness of Resolution

Definitions

- a clause is called ground if it doesn't contain variables
- a ground substitution is a substitution whose range contains only terms without variables
- let $\square \notin \mathrm{Res}^*(\mathcal{C})$, then $\mathcal{C}$ is consistent

Lemma

- *let S denote the set of all consistent ground clause sets*
- *then S has the satisfaction properties*

Proof.

(sort of) homework ∎

Lifting Lemmas

Lemma

- *let τ_1 and τ_2 be a ground and consider*

$$\frac{C\tau_1 \vee A\tau_1 \quad D\tau_2 \vee \neg B\tau_2}{C\tau_1 \vee D\tau_2}$$

where $A\tau_1 = B\tau_2$

- *$\exists$ mgu σ of A and B, such that σ is more general then τ_1 and τ_2 and the following resolution step is valid:*

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

Lemma

- *let τ be a ground substitutions and consider the following ground factoring step:*

$$\frac{C\tau \vee A\tau \vee B\tau}{C\tau \vee A\tau}$$

where $A\tau = B\tau$

- *$\exists$ mgu σ, such that σ is more general then τ and the following resolution step is valid:*

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

Lemma

- *let τ be a ground substitutions and consider the following ground factoring step:*

$$\frac{C\tau \vee A\tau \vee B\tau}{C\tau \vee A\tau}$$

where $A\tau = B\tau$

- *$\exists$ mgu σ, such that σ is more general then τ and the following resolution step is valid:*

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}$$

Proof.

the lemmas essentially follows from the properties of an mgu ∎

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable
2. $\exists$ a set of ground clauses $\mathcal{C}'$ that are instances of the clauses in $\mathcal{C}$ such that $\mathcal{C}'$ is unsatisfiable

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable
2. $\exists$ a set of ground clauses $\mathcal{C}'$ that are instances of the clauses in $\mathcal{C}$ such that $\mathcal{C}'$ is unsatisfiable
3. suppose $\square \notin \mathrm{Res}^*(\mathcal{C}')$

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable
2. $\exists$ a set of ground clauses $\mathcal{C}'$ that are instances of the clauses in $\mathcal{C}$ such that $\mathcal{C}'$ is unsatisfiable
3. suppose $\square \notin \mathrm{Res}^*(\mathcal{C}')$
4. by definition $\mathcal{C}'$ is consistent

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable
2. $\exists$ a set of ground clauses $\mathcal{C}'$ that are instances of the clauses in $\mathcal{C}$ such that $\mathcal{C}'$ is unsatisfiable
3. suppose $\square \notin \mathrm{Res}^*(\mathcal{C}')$
4. by definition $\mathcal{C}'$ is consistent
5. by model existence $\mathcal{C}'$ is satisfiable

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\Box \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable
2. $\exists$ a set of ground clauses $\mathcal{C}'$ that are instances of the clauses in $\mathcal{C}$ such that $\mathcal{C}'$ is unsatisfiable
3. suppose $\Box \notin \mathrm{Res}^*(\mathcal{C}')$
4. by definition $\mathcal{C}'$ is consistent
5. by model existence $\mathcal{C}'$ is satisfiable
6. contradiction to our assumption, hence $\Box \in \mathrm{Res}^*(\mathcal{C}')$

Theorem

resolution is complete; if F a sentence and $\mathcal{C}$ its clause form, then $\square \in \mathrm{Res}^*(\mathcal{C})$ if F is unsatisfiable

Proof.

1. suppose F is unsatisfiable
2. $\exists$ a set of ground clauses $\mathcal{C}'$ that are instances of the clauses in $\mathcal{C}$ such that $\mathcal{C}'$ is unsatisfiable
3. suppose $\square \notin \mathrm{Res}^*(\mathcal{C}')$
4. by definition $\mathcal{C}'$ is consistent
5. by model existence $\mathcal{C}'$ is satisfiable
6. contradiction to our assumption, hence $\square \in \mathrm{Res}^*(\mathcal{C}')$
7. the lifting lemmas allows to lift this derivation to show $\square \in \mathrm{Res}^*(\mathcal{C})$

∎