

Introduction

Logics plays an essential rôle in Computer Science and in Artificial Intelligence. This will be demonstrated with some examples in this introduction.

Programming

Within the last decade it turned out that computerised systems are the very base of advanced technology. Software is present in nearly all devices of modern houses, in our cars, not to speak about aircrafts or weapons. Without going into details, it is immediately obvious that for most, if not all applications, robust, safe and correct behaviour of a system is mandatory. To achieve this it is widely accepted that it is only possible if formal methods are applied during the entire process of hard- and software-development. In the following we shortly depict some tasks where the use of logic has proved to be extremely helpful.

Abstract Datatypes In order to define datatypes and to derive efficient implementations for it the concept of abstract datatype definitions is central. The idea is to define the abstract properties of a datatype, instead of giving special realisations. A very trivial example is the definition of a *stack*:

Let Σ be an alphabet. In order to define a stack S over Σ we assume *clear* to be a nullary function and we define the following properties of stacks:

$$\begin{aligned} & \text{clear} \in S \\ & \forall s \in S \forall x \in \Sigma (push(s, x) \in S) \\ & \forall s \in S (s \neq \text{clear} \Rightarrow pop(s) \in S) \\ & \forall s \in S (s \neq \text{clear} \Rightarrow top(s) \in \Sigma) \\ & \forall s \in S (empty(s) \in \{true, false\}) \end{aligned}$$

Hence, we have the functions *clear*, *push* and *pop*, which yield stacks and the predicate *empty*, furthermore we need the following properties with respect to the *push*-operation.

$$\begin{aligned} & \forall s \in S \forall x \in \Sigma (push(s, x) \neq \text{clear}) \\ & \forall s \in S \forall x, y \in \Sigma (push(s, x) = push(s, y) \Rightarrow x = y) \\ & \forall s, t \in S \forall x \in \Sigma (push(s, x) = push(t, x) \Rightarrow s = t) \end{aligned}$$

And we have to give properties with respect of the combinations of functions:

$$\begin{aligned} & \forall s \in S \forall x \in \Sigma (top(push(s, x)) = x \wedge pop(push(s, x)) = s) \\ & \forall s \in S \forall x \in \Sigma (empty(push(s, x)) = false) \\ & \text{empty}(\text{clear}) = true \end{aligned}$$

The above formulae state what properties we expect stacks to have. Obviously it contains no hints how to implement such a data type. And indeed this specification is aiming to solve other problems, e.g.

- Is the specification correct? I.e. is there a set S together with the given operations, such that the axioms above hold?
- Is the specification complete. I.e. do the axioms imply all the properties we intuitively assume a stack to have? Are there sets S which do not meet our expectations?

The reader may have already noticed, that the axioms are nothing else then formulas in predicate logic, i.e. al logic where variables like x or s together with so called quantifiers \forall and \exists are used.

The above two questions, namely correctness and completeness, are very central topics for the design of formal systems in logics. The prove of these proprties often is difficult and costly, but on the other hand it is one of the clear advantages of logical systems, that these properties *can be proved formally*. In the main part of this course we will deal with these questions explicitly.

Program development There are a number of attempts to define methods, which allow the development of a program together with a formal argument of its correctness. We will give a very rough idea with a toy example.

Assume the following simple program containing a loop, and assume that a denotes an array of integers:

```

max := a(1);
do i = 2,n
  if a(i) > max then max := a(i)
end

```

In order to understand what is going on if this program is executed, the following so called *loop invariant* is helpful

$$\forall j : 1 \leq j \leq i \Rightarrow \text{max} \geq a(j)$$

This means, that for every value of i , i.e. before and after every execution of the if-statement within the do-loop the above formula is valid. Now assume that the loop is executed the last time, hence the value of i after executing of the loop-body is n , one can conclude that max contains the maximum value of the array:

$$\forall j : 1 \leq j \leq n \Rightarrow \text{max} \geq a(j)$$

Another important issue for program development is the **specification of programs**. In order to give a formal specification of the above program for finding the maximum of an array the following logical formula can be used.

$$\forall S \forall m \text{max}(S, m) \Leftrightarrow (S \neq \emptyset \Rightarrow (m \in S \wedge \forall x (x \in S \Rightarrow m \geq x)))$$

Note that in this specification S is assumed to be a set. There is no decision yet made, that this has to be implemented by means of an array.

On the other hand logic can be used not only for the specification but also as programs itself. The following is a logic program which computes the maximum value of a list of values. Lists are represented as [**head.tail**], where **head** denotes the front element of a list, **tail** the rest of the list and **nil** the empty list.

```

max([m.nil], m) <- .
max([head.tail], m) <- max([tail], m),
                        head < m.
max([head.tail], head) <- max([tail], m),
                        head >= m.

```

Artificial Intelligence

One of the oldest sub-disciplines of Artificial Intelligence (AI) research is automated theorem proving. In the early days some were very optimistic about using theorem provers as general problem solvers for various different tasks, like action planning, knowledge representation or program verification. Now it is clear, that for special tasks tailored reasoning systems are necessary. We will comment in this subsection on theorem proving, aiming at proving mathematical theorems and on knowledge representation. This idea can be seen as going back to the ideas of Gottfried Wilhelm Leibnitz (1646 - 1716), who already at this time had a dream of formalisation, and even automatization of mathematics.

Theorem Proving

A recent success is the proof of Robbins conjecture, which even reached the New York Times. For details see McCunes homepage.

In 1933, E. V. Huntington presented the following basis for Boolean algebra:

```

x + y = y + x.
                        [commutativity]
(x + y) + z = x + (y + z).
                        [associativity]
n(n(x) + y) + n(n(x) + n(y)) = x.
                        [Huntington equation]

```

Shortly thereafter, Herbert Robbins conjectured that the Huntington equation can be replaced with a simpler one:

```

n(n(x + y) + n(x + n(y))) = x.
                        [Robbins equation]

```

Robbins and Huntington could not find a proof, and the problem was later studied by Tarski and his students.

The proof that solves the Robbins problem was found October 10, 1996, by the theorem prover EQP. EQP is similar in many ways to the more well known program Otter. The main differences are that EQP has associative-commutative (AC) unification, is restricted to equational logic, and offers more paramodulation strategies. See the EQP preprint for details.

See also:

McCune
Otter
EQP preprint

Knowledge Representation In many Artificial Intelligence the representation and manipulation of knowledge is a central task. To this end numerous graphic-oriented formalisms have been invented. In figure ?? a small example is given.

An informal semantics of this graphical notation states, that both, *man* and *animal* are *mamal* and that *man* have a *nationality* and an *age*, whereas an *animal* has *age* and no *nationality*.

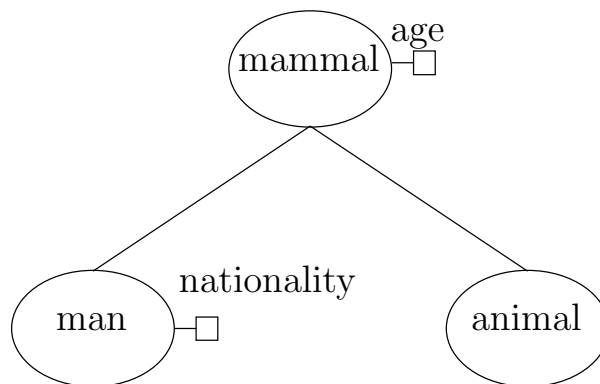
A closer investigation of the semantics of such a formalism would show, that this is nothing than a pictorial representation of the following set of predicate logic formulae:

$$\begin{aligned} \forall x(man(x) \Rightarrow mammal(x)) \\ \forall x(animal(x) \Rightarrow mammal(x)) \\ \forall x(man(x) \Rightarrow \exists y age(x, y)) \\ \forall x(man(x) \Rightarrow \exists y nationality(x, y)) \\ \forall x(mammal(x) \Rightarrow \exists y age(x)) \end{aligned}$$

See also:

Gellrich/Gellrich: Mathematik (1) - Schaltalgebra

Example



Problem 1

In a criminal case the following facts are proved:

1. At least one of the three persons X,Y,Z is guilty.
2. If X guilty and Y are innocent, then Z is guilty.

These circumstant are not sufficient to accuse one of them but it can be said for certain that at least one of two persons must be guilty. Which two are these?

Problem 2

Which of the following syllogisms are valid? Give a reason for your answer or give a counterexample.

1. All *M* are *P*, some *S* are not *M* then: Some *S* are *P*.
2. All *M* are *P*, some *S* are not *M* then: Some *S* are not *P*
3. All *P* are *M*, some *S* are not *M* then: Some *S* are not *P*.

Problem 3

In a meeting there are 100 politicians discussing with each other. Everyone of them is either corruptibly or uncorruptibly. Following facts are known:

1. at least a politician is uncorruptibly.
2. In each case of two politicians is at least one corruptibly .

How Many of the politicians are corruptibly, how many uncorruptibly?

Problem 4

The anthropologist Abercrombie entered the island of the knights and the mucker with a slack feeling he have never had before. He knew that very wondrous people lived on this island: The knights made only true propositions the mucker false propositions every time. Abercrombie knew also that he had to find a friend before he could experience something. He had to find someone whose propositions he can trust. So he asked the first three people of the island he met to find a knight. Aberbrombie asked Arthur at first: Are Bernard and Charles both knights? Arthur answered: Yes they are! Abercrombie asked then: Is Bernard a knight? With a big surprise he get the answer: No! Is Charles a knight or a mucker?(Raymond Smullyan)

Deduce your answer and give a reason for it.

Problem 5

A little island had exact 100 inhabitants. Every inhabitant said always the truth or lied always either. One Researcher came one day on the island and questioned the inhabitants sequentially. The first told: "There is at least a liar with us." The second said: "At least two liars live among us." etc.. The last finally claimed: "There are 100 liars on this Island." How many liars were there really?

The researcher went to another island with 99 Inhabitants a year later. In an interview the inhabitants of these island spoke completely corresponding like on the first island, i.e. the n th inhabitants said: "There are at least n liars here." What can you say about this island?

Problem 6

In a village the priest explained one Sunday: "It was confessed to me that there are men who are unfaithful in our village. However, the confessing secret forbids it to me to call the names. It will you nevertheless learn all of them, if we proceed as follows: Any woman who for certain knows that her husband is unfaithful shall throw him in the following night out of the house."

However, the problem was that every woman knows all about every other husband but nothing about her husband. In the next morning the priest went through the streets; not one single man was turned adrift. Also on the next day he saw nobody. But at 100th acre he saw men, who had been thrown out of the house by their wives. How many?

Problem 7

There is a hotel with countable infinitely many rooms $0, 1, 2, \dots$. All rooms are vacant. Now, a ω -decker bus comes with ω many seats on every deck. How can all of the passengers be accommodated in the hotel?

Induction

Induction plays a crucial rôle at least in two aspect throughout this book. Firstly, it is one of main proof principles in mathematics and of course in logics. In particular it can

be used to investigate properties of infinite sets. Very often it is used as *natural induction*, namely over the natural numbers. We will introduce it as a more general principle over well founded partial orders, which is called *structural induction*.

The second aspect is, that it can be used as well to define infinite structures, as the set of well formed formulae in a particular logic or the set of binary trees.

We start with a very general structure over arbitrary sets, namely partial orders.

A relation R over A is a partial order iff R is reflexive, transitive and anti-symmetric (i.e. $((x, y) \in R \wedge (y, x) \in R \Rightarrow x = y)$). Partial ordered sets (p.o. sets) A are usually written as (A, \leq) .

Definition 1 *The necessary structure for our induction principle is a partial order, such that there exist minimal elements. Given a p.o. set (A, \leq) , we define:*

- $x < y$ iff $x \leq y$ and $x \neq y$.
- (A, \leq) is called *well-founded* iff there is no infinite sequence $(x_i)_{i \in \mathbb{N}}$ and $x_{i+1} < x_i, \forall i \geq 0$.
- $X \subseteq A$ is called a *chain* iff $\forall x, y \in X : x \leq y$ or $y \leq x$
- (A, \leq) is a *total ordering* iff A is a chain.

Lemma 1 (A, \leq) is well-founded, iff every non-empty subset of A has a minimal Element.

Proof can be done by contradiction and will be given as an exercise.

We finally have the machinery to introduce the principle of complete induction:

Definition 2 (Complete (structural) induction) *Given a well-founded p.o. set (A, \leq) and a predicate P , i.e. $P : A \rightarrow \{true, false\}$. The principle of induction is given by the following (second order) formula*

$$\forall x \in A (\forall y \in A (y < x \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall z \in A P(z)$$

Lemma 2 *The induction principle holds for every well-founded set.*

Proof: The proof is given by contradiction: Assume the principle is wrong; i.e. the implication is wrong, which means, that we have to assume the premise as true:

$$\forall x \in A (\forall y \in A (y < x \Rightarrow P(y)) \Rightarrow P(x)) \equiv true \tag{1}$$

and the conclusion as wrong:

$$\forall z \in A (P(z)) \equiv false \tag{2}$$

Hence we can assume that the set $X = \{x \in A \mid P(x) = false\}$ is not empty. Since X is a subset of a well-founded set, it has a minimal element, say b . From assumption 1 we conclude

$$(\forall y \in A (y < b \Rightarrow P(y)) \Rightarrow P(b)) \equiv true \tag{3}$$

Now we can distinguish two cases:

- b is minimal in A : Hence there is no $y \in A$, such that $y < b$. Hence the premise $\forall y \in A (y < b \Rightarrow P(y))$ of the implication in 3 is true, which implies that the conclusion $P(b)$ is true. This is a contradiction to the assumption that $b \in X$!
- b is not minimal in A : $\forall y \in A (y < b)$ holds and it must be that $P(y)$ is true, because otherwise it would be that $y \in X$ and b not minimal in X . Hence, again the premise $\forall y \in A (y < b \Rightarrow P(y))$ of the implication in 3 is true, which implies that the conclusion $P(b)$ is true. This is a contradiction to the assumption that $b \in X$!

An Example

In this subsection we will carry out a proof with induction in detail. For this we need the extension of p.O. sets:

Definition 3 (Lexicographic Ordering) A p.O. set (A, \leq) induces an ordering \ll over $A \times A$: $\forall x, y, x', y' \in A : (x, y) \ll (x', y')$ iff

- $x = x'$ and $y = y'$ or
- $x < x'$ or
- $x = x'$ and $y < y'$

Lemma 3 If (A, \leq) is a well-founded set, then $(A \times A, \ll)$ is well-founded as well.

Theorem 1 The Ackermann-function ACK is defined by the following recursion is a total function over $N \times N$

$ACK(x, y) =$ if $x=0$ then $y+1$ else if $y=0$ then $ACK(x-1, 1)$ else $ACK(x-1, ACK(x, y-1))$

Proof:

For the induction start we take the minimal element $(0, 0)$ of the well-founded set, $(N \times N, \ll)$, where \ll is the lexicographic ordering induced by (N, \leq) . Hence, assume $x = 0, y = 0$. By definition of ACK , we conclude $ACK(0, 0) = 1$ and hence defined.

Assume for an arbitrary (m, n) , that

$ACK(m', n')$ is defined for all $(m', n') \ll (m, n)$, if $(m, n) \neq (m', n')$

We distinguish the following cases:

- $m = 0$: i.e. $ACK(0, n) = n + 1$ and hence defined.
- $m \neq 0$ and $n = 0$: We know that $(m - 1, 1) \ll (m, 0)$ and $(m - 1, 1) \neq (m, 0)$. From the induction hypothesis we know that $ACK(m - 1, 1)$ is defined, and hence $ACK(m, 0) = ACK(m - 1, 1)$ as well.
- $m \neq 0$ and $n \neq 0$: According to the definition of ACK we have two cases to consider:
 - $(m, n - 1) \ll (m, n)$ and $(m, n - 1) \neq (m, n)$: From the induction hypothesis we conclude immediately that $ACK(m, n - 1)$ is defined.

- $(m - 1, y) \ll (m, z)$ and $(m - 1, y) \neq (m, z)$: Independent from the values of x and y . If we assume $y'ACK(m, n - 1)$ and $z = n$, we again can conclude from the hypothesis, that $ACK(m - 1, ACK(m, n - 1))$ is defined and hence $ACK(m, n)$ as well

Altogether we proved, that $ACK(x, y)$ is defined for all $x, y \geq 0$.

Problem 1

Prove the following lemma: If (A, \leq) is well founded also $(A \times A, \ll)$.

Note: The *lexicographical Order* $(A \times A, \ll)$ is defined as follows:

$$(m, n) \ll (m', n') \iff (m < m' \vee (m = m' \wedge n \leq n'))$$

Problem 2

How many points of intersection could n straight lines have at most? Find a recursive and explicit formula and show

Problem 3

Prove that a number x is even if and only if x^2 is even.

Problem 4

Point by an indirect proof that there is not any greatest prime number!

Problem 5

Which prerequisite do you need that the following order

$(\{A, B, C, 0, 1, 2\}, \leq)$ is

1. partial ordered
2. total ordered
3. well-founded?

Problem 6

An example of a well founded set is the power set $P(M)$ over a finite set M which is comparable over the relation of the subset \subseteq . If $M = \{1, 2, 3\}$ then is e.g. $\{1\} \subseteq \{1, 2, 3\}$ but $\{1, 2\}$ and $\{2, 3\}$ are not comparable. Give a definition of a relation B in this way that $(P(M), B)$ is total ordered and well founded.

Problem 7

Examine which of the following partial order are total and which are well founded!

1. $(2^{\mathbb{N}}, \subseteq)$ with $2^{\mathbb{N}}$ is the power set for natural numbers.
2. $(\mathbb{N}, |)$ with $|$ marks the relation "is factor of".
3. $(\mathbb{N} \times \mathbb{N}, \ll)$ with $(m, n) \ll (m', n')$ iff $m < m'$ or $(m = m'$ and $n \leq n')$.
4. (\mathbb{N}^*, \preceq) with \preceq is the lexicographical .
5. for \mathbb{N}^* , i.e. $1 \prec 1.1 \prec 3 \prec 3.3.8$

Problem 8

Give for the natural numbers \mathbb{N} an order relation, that is

1. both well-founded and total,
2. total but not well-founded,

3. well-founded but not total and
4. neither well-founded nor total.

Problem 9

Prove, that a partial order (A, \leq) is well-founded iff every non-empty partial set of A (at least) contains a minimum element.

Problem 10

A root tree consists (a) of a single knot or (b) of a knot - that's the root of the tree - and at least one, but only at most finite many (part)trees, this one is bandaged over an edge with the root. Point formally by means of induction, that in every tree the number of the knots n around 1 is taller than the number of the edges e , i.e. $e = n - 1$.

Problem 11

Prove: If \mathcal{E} is a quality of the natural numbers \mathbb{N} and it is valid

1. $\mathcal{E}(0)$ and
2. $\forall n \in \mathbb{N} : [\mathcal{E}(n) \Rightarrow \mathcal{E}(n + 1)]$.

then $\forall n \in \mathbb{N} : \mathcal{E}(n)$ is valid.

Note: The proof can be done by the fact that the principle of the complete induction in \mathbb{N} (which should be proved) can be reduced to the principle of the transfinite induction for well founded orders.

Propositional Logic

This section introduces propositional logic. We will study syntax and model theoretic semantic of a language of classical propositional logic and we investigate various calculi for deciding certain properties of sentences in this language.

Preliminaries

As a running example consider the simple digital circuit in figure 1 consisting of an or-gate (*or1*) and two inverters (*inv1* and *inv2*). The system description is given by the following propositional formulae, where the formulae are labelled in order to facilitate recognition of corresponding parts in figure 1. The mnemonic used in the names within the formulae is the following:

- The symbols \vee , \wedge and \neg denote “or”, “and” and “not” respectively.
- *i1* and *i2* are denoting inputs in the or-gate, *i* an inverter and *o* denotes the output of a gate.
- *high* is a predicate which is intended to denote the fact, that the value specified by its arguments is “of high voltage”, i.e. is on.
- *ab* is a predicate which stands for “is abnormal”, i.e. the expression $\neg(ab(inv1))$ can be read as “it is not the case, that the inverter one is not abnormal.

$$\begin{aligned}
OR1 &: \neg(ab(or1)) \rightarrow high(or1, o) \leftrightarrow (high(or1, i1) \vee high(or1, i2)) \\
INV1 &: \neg(ab(inv1)) \rightarrow high(inv1, o) \leftrightarrow \neg(high(inv1, i)) \\
INV2 &: \neg(ab(inv2)) \rightarrow high(inv2, o) \leftrightarrow \neg(high(inv2, i)) \\
CONN1 &: high(inv1, o) \leftrightarrow high(or1, i1) \\
CONN2 &: high(inv2, o) \leftrightarrow high(or1, i2)
\end{aligned}$$

These formulae contain labels which express their intended meaning: OR1 stands for a formula describing the function of the or-gate, INV1 and INV2 are describing the (identical) behavior of the two inverters: CONN1 and CONN2 are specifying the way the inputs of the or-gate are connected with the outputs of the inverters. Note, that in this description the inputs and outputs of the system are not yet defined.

We observe from the graphical description that both inputs of the circuit have low voltage and the output also has low voltage, i.e. we could state this by the formulae $\neg high(inv1, i)$, $\neg high(inv2, i)$, $\neg high(or1, o)$

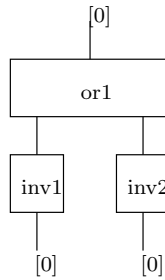


Figure 1: A simple electronic circuit

This example demonstrates how a proposition like $high(inv1, o)$ can be used to formally describe the behaviour of an electronic circuit. Note that $high(inv1, o)$ can be read intuitively as ‘‘the output of inverter 1 is high’’, and of course this sentence can be further simplified by ignoring the phrase structure and by understanding it as a simple string: ‘‘the_output_of_inverter_1_is_high’’. If we would proceed with the other propositions in the same manner we would get formulae like this:

$$the_output_of_inverter_1_is_high \leftrightarrow the_input_of_inverter_1_is_high$$

This formula consists of two propositions which are connected via a double arrow. It is easy to see that formulae get rather lengthy by this way and hence we can further abbreviate the propositions by simpler strings like P_1 or P_2 , which results in a formula $P_1 \leftrightarrow P_2$. It should be clear by the above argumentation that this formula carries for our purposes the same information as $high(or1, i1) \leftrightarrow high(inv1, o)$. The only difference is that with the latter form some intuition is associated with the reader – for our formal logical framework this is irrelevant and we will focus on arbitrary propositions.

In the following we will systematically avoid any intuition to real world meaning in the design of our language. We will focus solely on the study of the logical aspects, which can be expressed by our language and not to a certain intended meaning. In other words, our

previous introduction of the logical formulae, which described the toy electronic circuit was too much oriented on an interpretation, which was triggered by the picture. We will introduce the syntax of propositional logic by a very simple inductive definition and then we specify a formal semantics by another single definition.

Syntax

For the definition of the syntax we have to assume a set of atomic formulae, which we take as the start of an inductive definition. With the help of three junctors and brackets as punctuation symbols we inductively define more complex formulae.

Definition 4 (Syntax of propositional logic) *Assume a*

- *countable set of atomic formulae P_i , where $i = 1, 2, 3, \dots$,*
- *the junctors \wedge, \vee , and \neg ,*
- *the punctuation symbols (and).*

The set of propositional formulae is defined by the following induction:

- *Atomic formulae are formulae.*
- *If F and G are formulae, then $(F \wedge G)$ and $(F \vee G)$ are formulae.*
- *If F is a formula, then $\neg F$ is a formula*

The different kinds of formulae are called conjunction, disjunction and negation. Note, that this is just a convention about wording, until now, we didn't give any semantics to these junctors, which would justify these denotations.

This definition of the set of formulae also introduces a partial order in a very natural way, if we consider the concept of a subformula.

Definition 5 (Subformula) *A subformula of a formula is a substring, which is a formula.*

Note that the relation “is subformula” is indeed a partial order.

The set of formulae together with the subformula-relation is a well-founded relation.

In order to facilitate notations We introduce the following abbreviations:

A, B, C	instead of	P_1, P_2, P_3
$(F_1 \rightarrow F_2)$		$(\neg F_1 \vee F_2)$
$(F_1 \leftrightarrow F_2)$		$((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$
$\bigvee_{i=1}^n F_i$		$(\dots ((F_1 \vee F_2) \vee F_3) \vee \dots \vee F_n)$
$\bigwedge_{i=1}^n F_i$		$(\dots ((F_1 \wedge F_2) \wedge F_3) \wedge \dots \wedge F_n)$

These notations (except the first one) are simple abbreviations; whenever those arrow-symbols or indexed junctors occur in a formulae, the corresponding subformula can be substituted according to this definition.

Coming back to our previous example you can see that

$$\neg(\text{ab}(\text{or1})) \rightarrow \text{high}(\text{or1}, o) \leftrightarrow (\text{high}(\text{or1}, i1) \vee \text{high}(\text{or1}, i2))$$

can be written as a formula according to the above definition by introducing parenthesis:

$$(\neg(ab(or1)) \rightarrow (high(or1, o) \leftrightarrow (high(or1, i1) \vee high(or1, i2))))$$

The parenthesis are introduced in order to avoid any ambiguity. In order to increase readability, we will omit them whenever this is possible.

If we additionally apply the above abbreviation rules we would result in the following formula:

$$\begin{aligned} (\neg\neg(ab(or1)) \vee \\ (high(or1, o) \wedge ((high(or1, i1) \vee high(or1, i2)) \vee \\ (\neg high(or1, o) \vee \neg(high(or1, i1) \vee high(or1, i2))))) \end{aligned}$$

Semantics

Definition 6 (Semantics of propositional logic) *For the semantics of our formal language of propositions we do not refer to a specific intended interpretation. Moreover we only are interested in truth values. We assume an initial assignment of truth values to atomic formulae and based on this, we define the truth value of more complex formulae. The set of truth values is the set $\{true, false\}$.*

An assignment for a set D of atomic formulae is a function $\mathcal{A}|D \rightarrow \{true, false\}$.

Let E be the set of formulae containing D , i.e. $E \supseteq D$ and exactly those formulae which can be constructed from D according to the definition of the syntax. Then the extension of \mathcal{A} on E , $\mathcal{A}_E|E \rightarrow \{true, false\}$, is given as:

- $A \in D : \mathcal{A}_E(A) = \mathcal{A}(A)$
- $\mathcal{A}_E((F \wedge G)) = \begin{cases} true & \text{if } \mathcal{A}_E(F) = true \text{ and } \mathcal{A}_E(G) = true \\ false & \text{otherwise} \end{cases}$
- $\mathcal{A}_E((F \vee G)) = \begin{cases} true & \text{if } \mathcal{A}_E(F) = true \text{ or } \mathcal{A}_E(G) = true \\ false & \text{otherwise} \end{cases}$
- $\mathcal{A}_E(\neg F) = \begin{cases} true & \text{if } \mathcal{A}_E(F) = false \\ false & \text{otherwise} \end{cases}$

In the following we will omit the index E to indicate the extension of an assignment \mathcal{A} . Note that this is the right place to name formulae of type $(F \wedge G)$ as conjunctions, formulae of type $(F \vee G)$ as disjunctions and formulae of type $\neg F$ as negations. The following is an example evaluation by means of the definition of \mathcal{A} :

Assume as given $\mathcal{A}(A) = true$, $\mathcal{A}(B) = true$ and $\mathcal{A}(C) = false$, then

$$\begin{aligned}
\mathcal{A}(\neg((A \wedge B) \vee C)) &= \begin{cases} true & \text{if } \mathcal{A}(((A \wedge B)) \vee C) = false \\ false & \text{otherwise} \end{cases} \\
&= \begin{cases} false & \text{if } \mathcal{A}(((A \wedge B)) \vee C) = true \\ true & \text{otherwise} \end{cases} \\
&= \begin{cases} false & \text{if } \mathcal{A}(((A \wedge B))) = true \text{ or } \mathcal{A}(C) = true \\ true & \text{otherwise} \end{cases} \\
&= \begin{cases} false & \text{if } \mathcal{A}(((A \wedge B))) = true \\ true & \text{otherwise} \end{cases} \\
&= \begin{cases} false & \text{if } \mathcal{A}(A) = true \text{ and } \mathcal{A}(B) = true \\ true & \text{otherwise} \end{cases} \\
&= false
\end{aligned}$$

Definition 7 Let \mathcal{A} be an assignment and F a formula. \mathcal{A} is called assignment for F , if \mathcal{A} is defined for every subformula of F .

If \mathcal{A} is an assignment for F and $\mathcal{A}(F) = true$ we call \mathcal{A} a model for F and we write $\mathcal{A} \models F$

If \mathcal{A} is an assignment for F and $\mathcal{A}(F) = false$, we write $\mathcal{A} \not\models F$.

A formula F is called satisfiable, iff there is a model for F , otherwise F is called unsatisfiable. F is called valid (or a tautology) iff every assignment for F is a model. We write $\models F$ resp. $\not\models F$.

Theorem 2 A formula F is valid, iff $\neg F$ is unsatisfiable.

Proof:

F is a tautology

iff every assignment for F is a model for F

iff every assignment for F (which is of course, also an assignment for $\neg F$) is not a model for $\neg F$

iff $\neg F$ has no model

iff $\neg F$ is unsatisfiable.

Definition 8 (Consequence) A formula G is called a consequence of the formulae F_1, \dots, F_k , if for every assignment \mathcal{A} for G and F_1, \dots, F_k holds: if \mathcal{A} is a model for F_1, \dots, F_k , then \mathcal{A} is a model for G . We write $\{F_1, \dots, F_k\} \models G$.

The following theorem is a simple consequence from definitions:

Theorem 3 G is called a consequence of the formulae F_1, \dots, F_k ,

iff $((\bigwedge_{i=1}^k F_i) \rightarrow G)$ is a tautologie

iff $((\bigwedge_{i=1}^k F_i) \wedge \neg G)$ is unsatisfiable.

Obviously the validity of a formula depends only of the assignments for its atomic subformulae: If \mathcal{A} and \mathcal{A}' are assignments for F and if they yield the same value for every occurring atomic subformulae, then $\mathcal{A}(F) = \mathcal{A}'(F)$ holds. As a consequence we can state, that it suffices for a test of the validity of a formula F to check the infinitely many assignments of its atomic subformulae.

Such a check can be depicted easily in a tableau of the following form, where F is an arbitrary formula, containing n distinct atomic formulae A_i .

	A_1	\dots	A_{n-1}	A_n	F
\mathcal{A}_1	false	\dots	false	false	$\mathcal{A}_1(F)$
\mathcal{A}_2	false	\dots	false	true	$\mathcal{A}_2(F)$
\mathcal{A}_{2n}	true	\dots	true	true	$\mathcal{A}_n(F)$

When applying this procedure it might be helpful to introduce intermediate results for subformulae, as done in the following example.

A	B	$\neg A$	$A \rightarrow B$	$(\neg A \rightarrow (A \rightarrow B))$
false	false	true	true	true
false	true	true	true	true
true	false	false	false	true
true	true	false	true	true

Note that we just have depicted an algorithm to check the validity of a formula. Assume the Formula contains n atomic subformulae, then our just constructed tableaux contain 2^n lines. To estimate the costs for such an exponential algorithm, assume that the computation of one line takes 1 micro-second. If F contains only 100 atomic subformulae the computation of the tableau would take 2^{100} micro-seconds.

Interaction: Truth Tables

This is your set of current propositional formulas:

Currently empty.

Current formulas manipulation:

<p>Add random formulas of complexity low medium high</p>	<p>Enter some formulas: Help:</p> <p>Add to / Overwrite current formulas</p> <p>Delete selected / Delete all current formulas</p>
--	--

Save/Reload current formulas:

<p>Save to database (enter name):</p>	<p>Reload from database (select name):</p> <p>Reload from file (enter file name):</p>
---	---



Generate truth table(s) for selected formulas Delete current truth tables

And here are the resulting truth tables (possibly from a previous run): Currently empty.
□

The problem whether a propositional formula is satisfiable is called SAT and the corresponding tautology problem is called TAUT.

SAT is an NP-complete problem, and hence it is not known whether SAT is tractable or not. Whether TAUT is in NP is still an open problem. For TAUT we know, that it is in NP iff NP is closed under complementation. SAT and TAUT, both, play a prominent role in the study of complexity theory, in particular with respect to the question whether $P = NP$.

Problem 1

Compute truth tables for the following formulae. Decide for each formula whether it is valid (a tautology), satisfiable or unsatisfiable.

1. $(A \wedge B) \wedge (B \rightarrow C)$
2. $(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
3. $(\neg A \vee \neg(\neg B \vee A)) \vee A$
4. $(A \rightarrow (B \wedge C)) \leftrightarrow ((A \rightarrow B) \wedge (A \rightarrow C))$
5. $(A \vee (B \wedge A)) \wedge ((C \vee \neg C) \rightarrow \neg A)$
6. $((C \vee B) \wedge (C \vee \neg B)) \wedge \neg C$
7. $\neg(\neg A \vee \neg(\neg B \vee \neg A))$
8. $(A \rightarrow (B \rightarrow A))$
9. $(A \vee (B \wedge A)) \wedge ((C \vee \neg C) \rightarrow \neg A)$
10. $((A \vee \neg B) \vee ((\neg A \wedge \neg C) \wedge B)) \leftrightarrow ((A \vee \neg C) \vee \neg B)$
11. $\neg(A \wedge \neg(B \wedge \neg C))$

Problem 2

"What is the secret of your long life?" a 100-year old man was asked. He answered: "I apply the following diet rules very strictly: If I drink no wine at dinner then I have always fish. Whenever I take fish and wine for dinner, it is without garlic. If I have garlic or wine I avoid fish"

1. Formalize these rules with the help of propositional logic.
2. Try to simplify the advice of the 100-year old man.

Problem 3

Define the following functions recursively by induction over the construction of propositional formulae:

1. $\alpha(F)$: Set of atomic formula in F
2. $\beta(F)$: Number of the binary junctors \wedge and \vee in F

Note: For

$$F = (\neg(A \vee B) \vee C) \wedge ((\neg A \vee B) \wedge C)$$

$\alpha(F) = \{A, B, C\}$ and $\beta(F) = 5$ holds.

Problem 4

Given the formulae $F_n = A_1 \dot{\vee} \cdots \dot{\vee} A_n$, in which A_1, \dots, A_n are atomic formulae ($n \geq 1$) where $\dot{\vee}$ denotes exclusive or. Prove for all $n \in \mathbb{N}$: A suitable assignment \mathcal{A} for F_n is a model for F_n (i.e. $\mathcal{A} \models F_n$) iff $\mathcal{A}(A_i) = 1$ holds for an odd number of A_i ($1 \leq i \leq n$).

Problem 5

In the following we investigate formulae, in which only atoms A_1, \dots, A_n occur.

1. How many of such formulae with different truth tables exist at most?
2. Is there for every truth table a formula? If yes, please indicate a construction!

Problem 6

If the colonel was not in the room during the murder then he cannot know the weapon of the murderer. The butler lies or he knows the murderer. If Lady Barntree is the murderer then the colonel was in the room during the murder or the butler lies. The butler knows the murderer or the colonel was not in the room during the murder. The policeman concludes that Lady Barntree is the murderer.

Give propositional variables for every statement of the argumentation. Write the argumentation as a set M of propositional formulae of the prerequisites and the conclusion as a propositional formula F .

Problem 7

Formalize the following expressions and then simplify the corresponding formulae and the verbal propositions.

1. It is not true that his mother is English and his father French.
2. It is not true that he is studying physics but not mathematics.
3. It is not true that the wages are going down and the prices are going up.
4. It is not true that it is not cold or rainy.

Problem 8

The professor proposes to make a new conception for the lunch in the University restaurant:

1. There must be bread with every lunch if there is no dessert.
2. If bread and dessert are served then there is no soup.
3. If soup is served then there is no bread.

Help the management to fulfill the wishes of the professor. For this

1. formalize the three propositions (as implications, disjunctions/conjunctions) and combine them into **one** logical formula.
2. Give a truth table for this logical formula. Is there a model for the formula?
3. Give a colloquial verbalization for the assignment.

Equivalence and Normal Forms

Until now, we only discussed single formulae and their semantical properties. In this section we start investigating whether formulae can be transformed into another form, without changing their semantics. For this we introduce the concept of *logical equivalence*, which can be used to investigate the transformation of a given formula into its *normal form*.

Definition 9 *The formulae F and G are called (semantically) equivalent, iff for all assignments \mathcal{A} for F and G , $\mathcal{A}(F) = \mathcal{A}(G)$. We write $F \equiv G$.*

Formulae containing different subformulae can be equivalent, e.g. all tautologies are equivalent.

More interesting is the following theorem:

Theorem 4 (Substitutivity) *Let $F \equiv G$ and H a formula which contains at least one occurrence of the subformula F . Then it holds $H \equiv H'$, where H' is obtained from H by substituting any occurrence of F by G .*

Proof: The proof is by induction over the structure of the subformula H :

Assume for the induction start, that H is atomic, hence $H = F$ holds, and the result of substituting the only occurrence of F by G results in $H' = G$ and because $F \equiv G$, we have $H \equiv H'$.

Assume the theorem holds for all proper subformulae of H : If $F = H$ we have the same argumentation as above in the start of the induction. If $F \neq H$ we have three cases:

- $H = \neg H_1$: Because H_1 is a subformula of H we can conclude that $H_1 \equiv H'_1$, where H'_1 is constructed by substituting any occurrence of F by G . From the definition of the semantics of \neg we conclude that $\neg H_1 \equiv \neg H'_1$ and hence $H \equiv H'$.
- $H = (H_1 \vee H_2)$: Assume without loss of generality, that F occurs in H_1 . Then, again we can conclude from the induction assumption, that $H_1 \equiv H'_1$ and from the definition of the semantics of \vee we conclude, that $(H_1 \vee H_2) \equiv (H'_1 \vee H_2)$.
- $H = (H_1 \wedge H_2)$ is similar.

Theorem 5 *The following equivalences hold:*

$$\begin{aligned}
(F \wedge F) &\equiv F \\
(F \vee F) &\equiv F && \text{(Idempotence)} \\
\\
(F \wedge G) &\equiv (G \wedge F) \\
(F \vee G) &\equiv (G \vee F) && \text{(Commutativity)} \\
\\
((F \wedge G) \wedge H) &\equiv (F \wedge (G \wedge H)) \\
((F \vee G) \vee H) &\equiv (F \vee (G \vee H)) && \text{(Associativity)} \\
\\
(F \wedge (F \vee G)) &\equiv F \\
(F \vee (F \wedge G)) &\equiv F && \text{(Absorption)} \\
\\
(F \wedge (G \vee H)) &\equiv ((F \wedge G) \vee (F \wedge H)) \\
(F \vee (G \wedge H)) &\equiv ((F \vee G) \wedge (F \vee H)) && \text{(Distributivity)} \\
\\
\neg \neg F &\equiv F && \text{(Double negation)} \\
\\
\neg(F \wedge G) &\equiv (\neg F \vee \neg G) \\
\neg(F \vee G) &\equiv (\neg F \wedge \neg G) && \text{(deMorgan's rule)} \\
\\
(F \vee G) &\equiv F, \text{ if } F \text{ is a tautology} \\
(F \wedge G) &\equiv G, \text{ if } F \text{ is a tautology} && \text{(Rule of Tautology)} \\
\\
(F \vee G) &\equiv G, \text{ if } F \text{ is unsatisfiable} \\
(F \wedge G) &\equiv F, \text{ if } F \text{ is unsatisfiable} && \text{(Rule of Unsatisfiability)}
\end{aligned}$$

Proof: All equivalences can be proved by truth tables. This is done here for the second rule of absorption:

F	G	$(F \wedge G)$	$(F \vee (F \wedge G))$
false	false	false	false
false	true	false	false
true	false	false	true
true	true	true	true

Let us now use these equivalences together with the theorem of substitutivity (TS) to prove the following equivalence:

$$((A \vee (B \vee C)) \wedge (C \vee \neg A)) \equiv ((B \wedge \neg A) \vee C)$$

$$((A \vee (B \vee C)) \wedge (C \vee \neg A))$$

$\equiv ((A \vee B) \vee C) \wedge (C \vee \neg A)$	Associativity and TS
$\equiv ((C \vee (A \vee B)) \wedge (C \vee \neg A))$	Commutativity and TS
$\equiv (C \vee ((A \vee B) \wedge \neg A))$	Distributivity
$\equiv (C \vee (\neg A \wedge (A \vee B)))$	Commutativity and TS
$\equiv (C \vee ((\neg A \wedge A) \vee (\neg A \wedge B)))$	Distributivity and TS
$\equiv (C \vee (\neg A \wedge B))$	Unsatisfiability and TS
$\equiv (C \vee (B \wedge \neg A))$	Commutativity and TS
$\equiv ((B \wedge \neg A) \vee C)$	Commutativity and TS

Problem 1

The binary junctor $\dot{\vee}$ for exclusive disjunction is defined by $F \dot{\vee} G = F \leftrightarrow (\neg G)$. Show that the following holds for propositional logic formulae F , G and H :

- $F \dot{\vee} G \equiv G \dot{\vee} F$ (Commutativity).
- $(F \dot{\vee} G) \dot{\vee} H \equiv F \dot{\vee} (G \dot{\vee} H)$ (Associativity).

Problem 2

Show the following by equivalence transformaton. Give to all remodelling steps the used rules!

- $(A \rightarrow (B \vee C)) \equiv (A \rightarrow B) \vee (A \rightarrow C)$
- $(A \rightarrow B) \rightarrow A \equiv (B \rightarrow A) \wedge (B \vee A)$
- $A \leftrightarrow \neg B \equiv \neg(A \leftrightarrow B)$
- $(A \rightarrow (B \wedge C)) \equiv (A \rightarrow B) \wedge (A \rightarrow C)$
- $(A \rightarrow B) \rightarrow (B \rightarrow A) \equiv B \rightarrow A$
- $(A \vee \neg B) \vee ((\neg A \wedge \neg C) \wedge B) \equiv (A \vee \neg C) \vee \neg B$

Problem 3

Prove with equivalences:

- $p \vee \neg(p \wedge q)$ is a tautology.
- $(p \vee \neg q) \wedge (\neg p \wedge q)$ is unsatisfiable.

Problem 4

The binary junctor \downarrow with the meaning *neither-nor* is defined by $F \downarrow G = \neg(F \vee G)$. Let F be a propositional logic formula, which contains only the operators \wedge , \vee and \neg . Prove that for every formula F a formula $\mathcal{T}(F)$ exists, such that $F \equiv \mathcal{T}(F)$ and $\mathcal{T}(F)$ is built by using only the junctor \downarrow .

Problem 5

Let F be a propositional logic formula, which contains only the operators \wedge , \vee and \neg . Prove that for every formula F a formula $\mathcal{T}(F)$ exists, such that $F \equiv \mathcal{T}(F)$ and $\mathcal{T}(F)$ is built by using only the junctors \rightarrow and $\dot{\vee}$.

Problem 6

The following formulae are given.

$$A \equiv D \vee B \quad B \equiv \neg B \vee \neg D \vee \neg S \quad C \equiv (\neg S \wedge D) \vee \neg B$$

1. Simplify $A \wedge B \wedge C$ by using $B \wedge C \equiv C$.
2. Simplify $A \wedge B \wedge C$ by using equivalences but not $B \wedge C \equiv C$.

Definition 10 (Normal Forms) A literal is an atomic formula or the negation of an atomic formula (positive or negative, resp.)

A formula F is in conjunctive normalform (CNF) iff

$$F = \left(\bigwedge_{i=1}^n \left(\bigvee_{j=1}^{m_i} L_{i,j} \right) \right)$$

where L_{ij} is a literal.

A formula F is in disjunctive normalform (DNF) iff

$$F = \left(\bigvee_{i=1}^n \left(\bigwedge_{j=1}^{m_i} L_{i,j} \right) \right)$$

where L_{ij} is a literal

Theorem 6 For every formula F there is an equivalent formula which is in DNF and an equivalent formula which is in CNF.

Let us formulate an algorithm to transform a given formula F into an equivalent normalform:

Given: A formula F

1. Substitute in F every subformula of the form

$$\begin{aligned} \neg \neg G & \text{ by } G \\ \neg (G \wedge H) & \text{ by } (\neg G \vee \neg H) \\ \neg (G \vee H) & \text{ by } (\neg G \wedge \neg H) \end{aligned}$$

until there is no subformula of this kind.

2. Substitute in the result from the above step every subformula of the form

$$\begin{aligned} (F \vee (G \wedge H)) & \text{ by } ((F \vee G) \wedge (F \vee H)) \\ ((F \wedge G) \vee H) & \text{ by } ((F \vee H) \wedge (G \vee H)) \end{aligned}$$

until there is no subformula of this kind.

Result: An equivalent formula in CNF

Interaction: Transformation into CNF

This is your set of current propositional formulas:

Currently empty.

Current formulas manipulation:

[Add](#) random formulas of complexity
low medium high

Enter some formulas: Help:

[Add to](#) / [Overwrite](#) current formulas


[Delete selected](#) / [Delete all](#) current formulas

Save/Reload current formulas:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):

[Convert selected](#) / [Convert all](#) formulas to CNF, replacing the current clauses. 

And this the resulting clause set (possibly from a previous run):

Currently empty.

Exercises

This is your set of current propositional formulas:

Currently empty.

Current formulas manipulation:

[Add](#) random formulas of complexity
low medium high

Enter some formulas: Help:

[Add to](#) / [Overwrite](#) current formulas

[Delete selected](#) / [Delete all](#) current formulas

Save/Reload current formulas:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):

Perform this exercise with the selected propositional formula

Current exercise:

Transform the following formula into CNF:

Currently empty. How to do the exercise:

Current transformations:

Currently empty.

[Apply](#) next formula:

[Help](#):

Until now, we investigate the transformation of a propositional formula into an equivalent normal form. Another problem in the context of normal forms is, to construct a normal form formula from a given truth table; i.e. the formula itself is not known, but its behaviour is given by a truth table. Let's read a normalform formula from a truth table: Assume a formula F , which is given by the following truth table.



A	B	C	F
false	false	false	true
false	false	true	false
false	true	false	false
false	true	true	false
true	false	false	true
true	false	true	true
true	true	false	false
true	true	true	false

In order to construct a formula in DNF, which is equivalent to F , we have to take into account, that every line of the table which yields the truthvalue *true* gives one conjunction: if the assignment of the literal A_i is *true* it is included as $\dots \wedge A_i \wedge \dots$, if is *false* we include $\dots \wedge \neg A_i \wedge \dots$. For the above example we get as a DNF:

$$\begin{aligned} &(\neg A \wedge \neg B \wedge \neg C) \vee \\ &(A \wedge \neg B \wedge \neg C) \vee \\ &(A \wedge \neg B \wedge C) \end{aligned}$$

If we change in the above procedure the roles of *true* and *false* and \vee and \wedge we arrive at a CNF:

$$\begin{aligned} &(A \vee B \vee \neg C) \wedge \\ &(A \vee \neg B \vee C) \wedge \\ &(A \vee \neg B \vee \neg C) \wedge \\ &(\neg A \vee \neg B \vee C) \wedge \\ &(\neg A \vee \neg B \vee \neg C) \end{aligned}$$

We introduce a special representation for formulae in normalform. In our circuit-example from the introduction we already used a very special form of normalforms, namely the implication form for formulae in CNF: every subformula F_i of a conjunction F_1, \dots, F_l is written as:

$$A_1 \wedge \dots \wedge A_n \rightarrow B_1 \vee \dots \vee B_m$$

It is easy to see, that this implication is logically equivalent to a disjunction $\neg A_1 \vee \dots \vee \neg A_n \vee B_1 \vee \dots \vee B_m$. Sometimes the implication is written as

$$A_1, \dots, A_n \rightarrow B_1; \dots; B_m$$

Even the following ambiguous notation is used in some cases, where the comma in the premiss stands for a conjunction and the comma in the conclusion for a disjunction:

$$A_1, \dots, A_n \rightarrow B_1, \dots, B_m$$

For some important procedures for logical reasoning it is mandatory to represent the formulae not only in one of the above notations for a normalform, moreover, it is necessary to use the so-called clause-form from the following definition.

Definition 11 *If F is a formula in CNF, i.e.*

$$F = (L_{1,1} \vee \dots \vee L_{1,n_1}) \wedge \dots \wedge (L_{k,1} \vee \dots \vee L_{k,n_k})$$

then its corresponding representation in clause form is given as

$$F = \{\{L_{1,1}, \dots, L_{1,n_1}\}, \dots, \{L_{k,1}, \dots, L_{k,n_k}\}\}$$

The sets $\{L_{i,1}, \dots, L_{i,n_i}\}$ are called clauses.

This representation as sets of literals has the advantage that literals occur in no special order and that multiple occurrences of a literal in a disjunction are “merged” in its clause form.

Note that as a consequence we have built in associativity, commutativity and Idempotence into the representation.

$$\begin{array}{ll} ((A_1 \vee \neg A_2) \wedge (A_3 \wedge A_3)) & \{\{A_1, \neg A_2\}, \{A_3\}\} \\ (A_3 \wedge (\neg A_2 \vee A_1)) & \\ \vdots & \end{array}$$

Problem 1

Create formulae in (a) conjunctive or (b) disjunctive normal form which are equivalent to:

$$A \dot{\vee} B \dot{\vee} C$$

where $\dot{\vee}$ denotes exclusive or.

Problem 2

The theorem prover OTTER uses the following optimization rules for clause sets:

Subsumption: If the literals of a clause K are a subset of an another clause K' remove K' from the set of clauses.

Deleting by unit clause: If the set of clauses contains a unit clause L -here L is single literal unit-clause- every occurrence of a complementary literal \bar{L} in a clause is deleted.

Which laws of the logic justify these procedures?

Problem 3

Generate truth tables for the following formulae. Give the conjunctive and disjunctive normal forms of the formulae. Which one of the relations $F \models G$, $G \models F$, $F \equiv G$ and $F = G$ holds?

1. $F = (A \wedge \neg(B \vee C)) \dot{\vee} ((A \rightarrow B) \vee (A \rightarrow C),)$ wobei $F \dot{\vee} G = F \leftrightarrow (\neg G)$ gilt.
2. $G = ((A \vee (B \wedge A)) \rightarrow \neg A) \vee C$

Problem 4

Generate a CNF and a DNF form the following truth table for the formula F .

A	B	C	F
0	0	0	0
1	0	0	1
0	1	0	0
1	0	0	1
1	1	0	1
1	0	1	0
0	1	1	1
1	1	1	0

Problem 5

Generate a CNF from the formula $(P \wedge (Q \rightarrow R) \rightarrow S)$

Horn clauses

In this subsection we introduce a special class of formulae which are of particular interest for logic programming. Furthermore it turns out that these formulae admit an efficient test for satisfiability.

Definition 12 *A formula is a Horn formula if it is in CNF and every disjunction contains at most one positive literal. Horn clauses are clauses, which contain at most one positive literal.*

$$\begin{aligned}
 F = & (A \vee \neg B) && \wedge && (\neg C \vee \neg A \vee D) \\
 & \wedge (\neg A \vee \neg B) && \wedge && D \wedge \neg E \\
 F \equiv & (B \rightarrow A) && \wedge && (C \wedge A \rightarrow D) \\
 & \wedge (A \wedge B \rightarrow \text{false}) && \wedge && (\text{true} \rightarrow D) \wedge (E \rightarrow \text{false})
 \end{aligned}$$

where *true* is a tautology and *false* is an unsatisfiable formula.
 In clause form this can be written as

$$\{A, \neg B\}, \{\neg C, \neg A, D\}, \{(\neg A, \neg B)\}, \{D\}, \{\neg E\}$$

and in the context of logic programming this is written as:

$$\left\{ \begin{array}{l} A \leftarrow B \\ D \leftarrow A \\ \quad \leftarrow A, B \\ \quad \leftarrow E \\ D \leftarrow \end{array} \right\}$$

For Horn formula there is an efficient algorithm to test satisfiability of a formula F :

Deciding Satisfiability of Horn Formulae

1. If there is a subformula of the kind $true \rightarrow A$ label every occurrence of A in F .
2. Apply the following rules until none of them is applicable:
 - If $A_1 \wedge \dots \wedge A_n \rightarrow B$ is a subformula and $A_1 \dots A_n$ are all labeled and B is not labeled then label every occurrence of B in F .
 - If $A_1 \wedge \dots \wedge A_n \rightarrow false$ is a subformula and $A_1 \dots A_n$ are all labeled then **Stop: Unsatisfiable**
3. **Stop: Satisfiable** The assignment $\mathcal{A}(A) = true$ iff A is labeled is a model.

Theorem 7 *The above algorithm is correct and stops after n steps, where n is the number of atoms in the formula.*

As an immediate consequence we see, that a Horn formula is satisfiable if there is no subformula of the form $A_1 \wedge \dots \wedge A_n \rightarrow false$.

Horn formulae admit least models, i.e. \mathcal{A} is a least model for F if for every model \mathcal{A}' and for every atom B in F holds: if $\mathcal{A}'(B) = true$ then $\mathcal{A}(B) = true$. Note, that this least model property does not hold for non Horn formulae: as an example take $A \vee B$ which is obviously non Horn. $\mathcal{A}_1(A) = true, \mathcal{A}_1(B) = false$ is a least model and $\mathcal{A}_2(A) = false, \mathcal{A}_2(B) = true$ as well, hence we have two least models.

Labeling algorithm

This is your set of current propositional clauses:

Currently empty.

Current clauses manipulation:

Enter some clauses:	Help:	Delete selected / Delete all current Clauses
Add to / Overwrite current clauses		Convert selected / Convert all current propositional formulas to CNF, replacing the current clauses.
Save/Reload current clauses:		
Save to database (enter name):		Reload from database (select name):
		Reload from file (enter file name):

[Apply labeling algorithm to the current clauses](#)

[Delete current labeling algorithm result](#)

This is the result, (possibly from a previous run):

Currently none. □

Problem 1

Let F be a propositional logical formulae and S a subset atomic formula occurring in F . Let $T_S(F)$ be the formula which results from F by replacing all occurrences of an atomic formulae $A \in S$ by $\neg A$. Example: $T_{\{A\}}(A \wedge B) = \neg A \wedge B$.

Prove or disprove: There exists an S for

1. $F = A \dot{\vee} B$ or
2. $F = A \dot{\vee} B \dot{\vee} C$,

so that $T_S(F)$ is equivalent to a Horn formula H (i.e. $T_S(F) \equiv H$).

Problem 2

Apply the marking algorithm to the following formula F . Which is a least model?

$$(A \vee \neg D \vee \neg E) \wedge (B \vee \neg C) \wedge (\neg B \vee \neg D) \wedge D \wedge (\neg D \vee E)$$

1. $F = (\neg A \vee \neg B \vee \neg C) \wedge \neg D \wedge (\neg E \vee A) \wedge E \wedge B \wedge (\neg F \vee C) \wedge F$
2. $F = (A \vee \neg D \vee \neg E) \wedge (B \vee \neg C) \wedge (\neg B \vee \neg D) \wedge D \wedge (\neg D \vee E)$

Problem 3

Decide which one of the indicated CNFs are Horn formulae and transform then into a conjunction of implications:

1. $(A \vee B \vee C) \wedge (A \vee \neg C) \wedge (\neg A \vee B) \wedge \neg B$
2. $(S \vee \neg P \vee Q) \wedge (S \vee \neg P \vee \neg R)$
3. $(A \vee \neg A)$
4. $A \wedge (\neg A \vee B) \wedge (\neg B \vee \neg C \vee D) \wedge (\neg E) \wedge (\neg A \vee \neg C) \wedge D$

Resolution

In this subsection we will develop a calculus for propositional logic. Until now we have a language, i.e. a set of formulae and we have investigated into semantics and some properties of formulae or sets of clauses. Now we will introduce an inference rule, namely the resolution rule, which allows to derive new clauses from given ones.

Definition 13 A clause R is a resolvent of clauses C_1 and C_2 , iff there is a literal $L \in C_1$ and $\bar{L} \in C_2$ and

$$R = (C_1 - \{L\}) \cup (C_2 - \{\bar{L}\})$$

$$\text{where } \bar{L} = \begin{cases} \neg A & \text{if } L = A \\ A & \text{if } L = \neg A \end{cases}$$

Note that there is a special case, if we construct the resolvent out of two literals, i.e. L and \bar{L} can be resolved upon and yield the empty set. This empty resolvent is depicted by the special symbol \square .

\square denotes an unsatisfiable formula. We define a clause set, which contains this empty clause to be unsatisfiable,

In the following we investigate in properties of the resolution rule and the entire calculus. The following Lemma is stating the correctness of one single application of the resolution rule.

Theorem 8 If S is a set of clauses and R a resolvent of $C_1, C_2 \in S$, then

$$S \equiv S \cup \{R\}$$

Proof: Let \mathcal{A} be an assignment for S ; hence it is an assignment for $S \equiv S \cup \{R\}$ as well. Assume $\mathcal{A} \models S$: hence for all clauses C from S , we have that $\mathcal{A} \models C$. The resolvent R of C_1 and C_2 looks like:

$$R = (C_1 - \{L\}) \cup (C_2 - \{\bar{L}\})$$

where $L \in C_1$ and $\bar{L} \in C_2$. Now there are two cases:

- $\mathcal{A} \models L$: From $\mathcal{A} \models C_2$ and $\mathcal{A} \not\models \bar{L}$, we conclude $\mathcal{A} \models (C_2 - \{\bar{L}\})$ and hence $\mathcal{A} \models R$.
- $\mathcal{A} \not\models L$: From $\mathcal{A} \models C_1$ we conclude $\mathcal{A} \models (C_1 - \{L\})$ and hence $\mathcal{A} \models R$.

The opposite direction of the lemma is obvious.

Definition 14 Let S be a set of clauses and

$$Res(S) = S \cup \{R \mid R \text{ is a resolvent of two clauses in } S\}$$

then

$$\begin{aligned} Res^0(S) &= S \\ Res^{n+1}(S) &= Res(Res^n(S)), n \geq 0 \\ Res^*(S) &= \bigcup_{n \geq 0} Res^n(S) \end{aligned}$$

If we understand the process of iterating the *Res*-operator as a procedure for deriving new clauses from a given set, and in particular to derive possibly the empty clause, we have to ask, under which circumstances we get the empty clause, and vice versa, what does it mean if we get it. These properties are investigated in the following two Theorems.

Theorem 9 (Correctness) *Let S be a set of clauses. If $\square \in Res^*(S)$ then S is unsatisfiable.*

Proof: From $\square \in Res^*(S)$ we conclude, that \square is obtained by resolution from two clauses $C_1 = \{L\}$ and $C_2 = \{\bar{L}\}$. Hence there is a $\exists n \geq 0$ such that $\square \in Res^n(S)$ and $C_1, C_2 \in Res^n(S)$ and therefore $Res^n(S)$ is unsatisfiable. From Theorem 8 we conclude that $Res^n(S) \equiv S$ and hence S is unsatisfiable.

Theorem 10 (Completeness) *Let S be a finite set of clauses. If S is unsatisfiable then $\square \in Res^*(S)$.*

Proof: Induction over the number n of atomic formulae in S .

With $n = 0$ we have $S = \{\square\}$ and hence $\square \in S \subseteq Res^*(S)$.

Assume n fixed and for every unsatisfiable set of clauses S with n atomic formulae A_1, \dots, A_n it holds that $\square \in Res^*(S)$.

Assume a set of clauses S with atomic formulae A_1, \dots, A_n, A_{n+1} . In the following we construct two clause sets S_f and S_t :

- S_f is received from S by deleting every occurrence of A_{n+1} in a clause and by deleting every clause which contains an occurrence of $\neg A_{n+1}$.
This transformation obviously corresponds to interpreting the atom A_{n+1} with *false*,
- S_t results from a similar transformation, where occurrences of $\neg A_{n+1}$ and clauses containing A_{n+1} are deleted, hence A_{n+1} is interpreted with *true*.

Let us show, that both S_f and S_t are unsatisfiable: Assume an assignment \mathcal{A} for the atomic formulae $\{A_1, \dots, A_n\}$ which is a model for S_f . Hence the assignment

$$\mathcal{A}'(B) = \begin{cases} \mathcal{A}(B) & \text{if } B \in \{A_1, \dots, A_n\} \\ \text{false} & \text{if } B = A_{n+1} \end{cases} \text{ is a model for } S, \text{ which leads to a contradiction.}$$

A similar construction shows that S_t is unsatisfiable. Hence we can use the induction assumption to conclude that $\square \in Res^*(S_t)$ and $\square \in Res^*(S_f)$.

Hence there is a sequence of clauses

$$C_1, \dots, C_m = \square$$

such that $\forall 1 \leq i \leq m$ it holds $C_i \in S_f$ or C_i is a resolvent of two clauses C_a and C_b with $a, b < i$. There is an analogous sequence for S_t :

$$C'_1, \dots, C'_t = \square$$

Now we are going to reintroduce the previously deleted literals A_{n+1} and $\neg A_{n+1}$ in the two sequences:

- Clause C_i which has been the result of deleting A_{n+1} from the original clause in S are again modified to $C_i \cup \{A_{n+1}\}$. This results in a sequence

$$\overline{C_1}, \dots, \overline{C_m}$$

where $\overline{C_m}$ is either \square or A_{n+1} .

- Analogous we introduce $\neg A_{n+1}$ in the second sequence, such that $\overline{C'_t}$ is either \square or $\neg A_{n+1}$

In any of the above cases we get $\square \in Res^*(S)$ latest after one resolution step with $\overline{C_m}$ and $\overline{C'_t}$.

Instantiated completeness proof

This is your set of current propositional clauses:

Currently empty.

Current clauses manipulation:

Enter some Clauses:

Help:

[Delete selected](#) / [Delete all](#) current Clauses

[Add to](#) / [Overwrite](#) current clauses

[Convert selected](#) / [Convert all](#) current propositional formulas to CNF, replacing the current clauses.

Save/Reload current clauses:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):

[Generate the resolution completeness of for the current clauses](#)  [Delete current proof](#)

This is your set S of current clauses : $\{\}$

Unfortunately S is not unsatisfiable, so it does not make sense to explain the proof with it. Please try again with an unsatisfiable set of clauses.

Based on the theorems for correctness and completeness, we give a procedure for deciding the satisfiability of propositional formulae.

Deciding Satisfiability of Propositional Formulae

Given a propositional formula F .

- Transform F into an equivalent CNF S .
- Compute $Res^n(S)$ for $n = 0, 1, 2, \dots$
 - If $\square \in Res^n(S)$ then **Stop: unsatisfiable**.
 - if $Res^n(S) = Res^{n+1}(S)$ then **Stop: satisfiable**.

Theorem 11 *If S is a finite set of clauses, then there exists a $k \geq 0$ such that*

$$Res^k(S) = Res^{k+1}(S)$$

Until now, we have been dealing with sets of clauses. In the following it will turn out, that it is helpful to talk about sequences of applications of the resolution rule.

Definition 15 *A deduction of a clause C from a set of clauses S is a sequence C_1, \dots, C_n , such that*

- $C_n = C$ and
- $\forall 1 \leq i \leq n : (C_i \in S \text{ or } \exists l, r < i : C_i \text{ is a resolvent of } C_l \text{ and } C_r)$

A deduction of the empty clause \square from S is called a refutation of S .

Example

We want to show, that the formula $K = ((B \wedge \neg A) \vee C)$ is a logical consequence of $F = ((A \vee (B \vee C)) \wedge (C \vee \neg A))$. For this negate K and prove the unsatisfiability of $F \wedge \neg K$

For this you can use the interaction in this book in various forms:

- Use the interaction *Truth Tables* for proving the unsatisfiability, or
- use the interaction *CNF Transformation* for transforming the formula into CNF, and then
- use the following interaction *Resolution*.

Current Propositional Clauses

This is your set of current propositional clauses:

Currently empty.

Current clauses manipulation:	
Enter some Clauses: <input style="width: 90%;" type="text"/> Help: <input style="width: 90%;" type="text"/> <u>Add to</u> / <u>Overwrite</u> current clauses	<u>Delete selected</u> / <u>Delete all</u> current Clauses <u>Convert selected</u> / <u>Convert all</u> current propositional formulas to CNF, replacing the current clauses. Save/Reload current clauses: <u>Save</u> to database (enter name): <input style="width: 90%;" type="text"/> <u>Reload</u> from database (select name): <input style="width: 90%;" type="text"/> <u>Reload</u> from file (enter file name): <input style="width: 90%;" type="text"/>



Interaction: Otter

Apply Otter to the current clauses Delete current Otter Result

This is the result, (possibly from a previous run):

Currently none. □

Interaction: Resolution Closure

Apply the “resolution closure” operator to the current set of clauses

Resulting set $Res^*(\text{current clauses})$ (possibly from a previous run):

Currently empty.

□

Delete current resolution closure

Problem 1

Compute $Res^n(M)$ for all $n \geq 0$ and $Res^*(M)$ for the following set of clauses:

1. $M = \{\{A\}, \{B\}, \{\neg A, C\}, \{B, \neg C, \neg D\}, \{\neg C, D\}, \{\neg D\}$
2. $M = \{\{A, \neg B\}, \{A, B\}, \{\neg A\}\}$
3. $M = \{\{A, B, C\}, \{\neg B, \neg C\}, \{\neg A, C\}\}$
4. $M = \{\{\neg A, \neg B\}, \{B, C\}, \{\neg C, A\}\}$

Which formula is satisfiable or which is unsatisfiable?

Problem 2

Indicate all resolvent of the clauses in S, where

$S = \{\{A, \neg B, C\}, \{A, B, E\}, \{\neg A, C, \neg D\}, \{A, \neg E\}\}$

Problem 3

Prove: A resolvent R of two clauses C_1 and C_2 is a logical consequence from C_1 and C_2 .

Note: Use the definition of "consequence".

Problem 4

Let M be a set of formulae and F a formula. Prove:

$$M \models F \text{ iff } M \cup \{\neg F\} \text{ is unsatisfiable.}$$

Problem 5

Compute $Res^n(S)$ with $n = 0, 1, 2$ and

$$S = \{\{A, \neg B, C\}, \{B, C\}, A\{\neg, C\}, \{B, \neg C\}, \{\neg C\}\}$$

Problem 6

Show that the following set S of formulae is unsatisfiable, by giving a refutation. $S = (B \vee C \vee D) \wedge (\neg C) \wedge (\neg B \vee C) \wedge (B \vee \neg D)$

Problem 7

Show by using the resolution rule, that $\neg A \wedge \neg B \wedge C$ is an inference from the set of clauses $F = \{\{A, C\}, \{\neg B, \neg C\}, \{\neg A\}\}$.

Problem 8

Show by using the resolution rule, that $((P \rightarrow Q) \wedge P) \rightarrow Q$ is a tautology.

Analytic Tableaux

In this section we present a calculus, namely analytic tableau, which is an alternative to resolution. Although this calculus was developed independently from resolution, it will turn out that there are some interesting common features. The most obvious difference is, that analytic tableau work direct on formulae, there is no need to transform a formula in a clausal normal form.

Tableaux - A Short History

The development of tableaux calculi started in the 1950th. The first authors to be mentioned are Beth (1955), Hintikka (1955) and Schütte (1956). Their goal was primarily to develop calculi without meta-language constructs. Later in the 1960th the idea of derivation trees and nodes in such a tree labeled by formulae became famous as the concept of *analytic tableaux* introduced by Smullyan 1968. The idea of mechanisation of tableaux calculi was then introduced by Kanger 1957, Prawitz 1960, Wang 1960, Davis 1960 and Maslov 1968.

Later on the concept of analytic tableau was modified and refined for its use in automated deduction by Loveland 1968 (Model elimination), Kowalski, Kuehner 1971 (SL-resolution) and Bibel 1975, Andrews 1976 (Connection or matings methods). Nowadays there are numerous high performance theorem provers based on this work.

The Calculus

One of the advantages of tableaux calculi is that can be defined without transforming the formula into clause normal form.

Example Given a set of formulae $\{\neg P \wedge \neg(Q \vee R), \neg(Q \wedge \neg R)\}$. We are aiming at constructing a tree, whose branches contain nodes which are labeled with formulae.

For this it is important to analyse the formula according to its leading connective. Smullyan observed that some work can be saved if non-literal formulas are grouped into types which are treated identically: α for formulas of conjunctive type, β for formulas of disjunctive type in the propositional case. Note that in the above example $\neg(Q \wedge \neg R)$ has

to be treated as a formula of disjunctive type, because of the negation standing in front of the conjunction.

Correspondence between formulas and their types is summarised in Table 1.

α	$\alpha_1, \dots, \alpha_n$	β	β_1, \dots, β_n
$\phi_1 \wedge \dots \wedge \phi_n$	ϕ_1, \dots, ϕ_n	$\phi_1 \vee \dots \vee \phi_n$	ϕ_1, \dots, ϕ_n
$\neg(\phi_1 \vee \dots \vee \phi_n)$	$\neg\phi_1, \dots, \neg\phi_n$	$\neg(\phi_1 \wedge \dots \wedge \phi_n)$	$\neg\phi_1, \dots, \neg\phi_n$
$\neg\neg\phi$	ϕ		
$\neg\text{false}$	true		
$\neg\text{true}$	false		

Table 1: Correspondence of formulas and their types.

The letters α and β are used to denote formulas of (and only of) the appropriate type. Let us now define the basic data structure for tableaux calculi together with the corresponding extension rules.

Definition 16 *A tableau for a logic L is a finitely branching tree whose nodes are formulas from L . A branch $branch$ in a tableau T is a maximal path in T . When no confusion can arise, branches are frequently identified with the set of their nodes (formulas). Given a set Φ of formulae from L , a tableau for Φ is constructed by a (possibly infinite) sequence of applications of the following rules:*

1. *The tree consisting of a single node true is a tableau for Φ (initialisation rule).*
2. *Let T be a tableau for Φ , B a branch of T , and ψ a formula in $B \cup \Phi$. If the tree T' is constructed by extending B by as many new linear subtrees as an instance of a tableau rule schema in Table 2 with premise ψ has extensions, and the nodes of the new subtrees are the formulas in the extensions of the rule instance, then T' is a tableau for Φ (expansion rule).*

α	β
α_1	$\beta_1 \mid \dots \mid \beta_n$
\vdots	
α_n	

Table 2: Rule schemata for tableaux.

One tableau for our example is depicted in figure 2.

Definition 17 *In a tableau T for a set Φ of sentences a branch B is closed iff $B \cup \Phi$ contains a pair $\phi, \neg\phi$ of complementary formulas, or false ; otherwise, it is open. A tableau is closed if all its branches are closed.*

A tableau proof for (the unsatisfiability of) a set Φ of formulae is a closed tableau T for Φ .

And here is the resulting tableau (possibly from a previous run):

Currently none. [Delete current tableau](#)

Problem 1

Give a strict tableau proof for the following formulae:

1. $\neg((A \downarrow B) \downarrow (A \vee B))$
2. $((A \rightarrow B) \wedge (B \rightarrow C)) \rightarrow (\neg(\neg C \wedge A))$

Clause Normalform Tableau

Let us now refine the calculus for the special case, that we deal with sets of clauses, which represent a formula in CNF. Note that in the case of conjunktive normal form clauses we only have literals, which are connected by \vee -junctors. Hence every clause is of β -type. In figure a tableau for a set of clauses is given. The clauses from the given clause set form the initial tableau; then there is only the β -rule applicable for further extensions of the tableau.

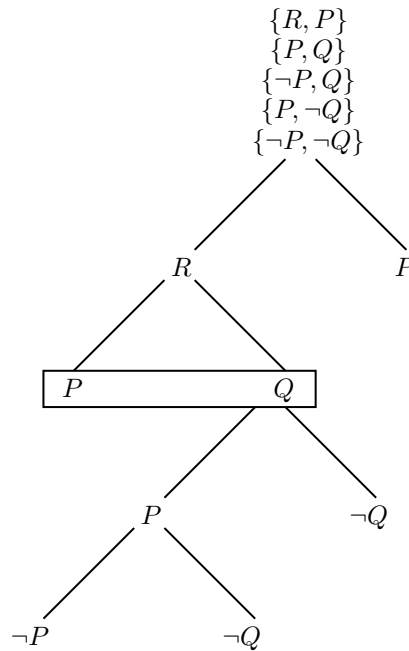


Figure 3: A tableau for clauses in CNF

In the following formal definition of a clause normal form tableau we start with an initial tableau, which is formed by taking an arbitrary clause from the given clause set S . For further extensions of the tableau β -rule-applications with clauses from S can be used. Which of the clauses is allowed is controlled by a *link condition*.

Definition 18 A clause (normalform) tableau for a set of clauses S is a tableau for S , whose nodes are literals from S and which is constructed by a (possibly infinite) sequence of applications of the following rules:

1. The tree consisting of root true and immediate successors L_1, \dots, L_n , where $C = L_1, \dots, L_n \in S$ is a tableau for S (initialisation rule).
2. Let T be a tableau for S , B a branch of T , and $C = L_1, \dots, L_n \in S$, such that the link-condition (see below) is satisfied. If the tree T' is constructed by extending B by the n subtrees L_i , then T' is a tableau for S (expansion rule).

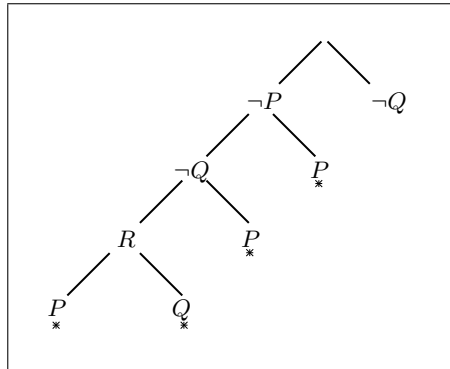
The following are three possible link conditions:

1. No condition.
2. Weak link condition: There is a literal $L \in B$ and $\bar{L} \in C$.
3. Strong link condition: Let L be the leaf of B , then there is $\bar{L} \in C$.

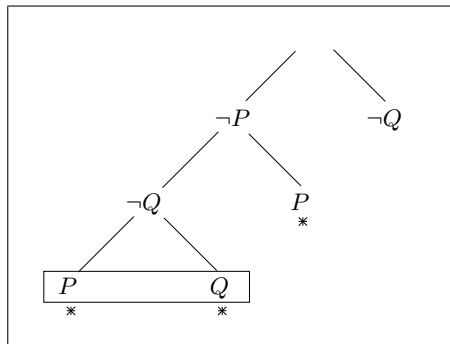
In fact we have defined three different calculi:

- Without link condition it is called clause normal form tableau,
- with the weak link condition we call it connection calculus, and
- with the strong link condition it is called model elimination.

An example for clause normal form tableau was given in figure). An example for a connection calculus tableau is



and finally a model elimination tableau is given by



Theorem 12 Clause normal form tableau are complete.

Note that strong link condition do not allow for confluent ¹ proof procedures. If no link condition (i.e. the empty one) is applied it is trivial to get a confluent version. For the case of weak link condition this is not obvious.

In order to arrive at a decision procedure for propositional clauses we need an extra condition:

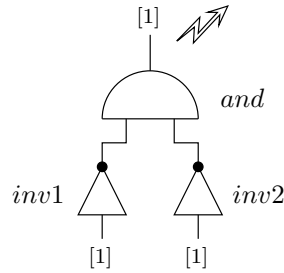
Definition 19 *A branch B of a tableau for a clause set S is called regular, if no literal occurs more than once.*

Theorem 13 *Clause normal form tableau with regularity and link condition give a decision procedure for propositional logic.*

¹ Let M be a set and \rightarrow a binary relation on M . Then (M, \rightarrow) is called *confluent* if

for all u, v, x, y and $u \rightarrow^* x$ and $u \rightarrow^* y$ there is a z with $x \rightarrow^* z$ and $y \rightarrow^* z$

Problem (diagnosis). Consider this electronic circuit with two input lines and one output line:



Suppose, as depicted, that both input lines are “1” and that the output line is “1”, thus contradicting the expected output value “0”.

1. First, formalize the circuit, i.e. the functionality of the three components and the two connections by neglecting the possibility of not correctly functioning components (i.e. do not use *abnormal*-Literals).
2. Consider the value pairs “0-0”, “1-0” and “1-1” to be supplied to the input lines. For each of these, using the result from (1) compute the expected output value by means of analytic tableau.

How did you read off the results from the tableaux?

3. Use your formalization and analytic tableau to prove that the output value “1” contradicts the expected behavior in case of input “1-1”.

How did you read off the result from the tableau?

4. Now modify the formalization of the components in (1) by *abnormal*-literals as shown in class.

Use analytic tableau to compute all possible diagnosis for the input “1-1” and output “1”.

How did you read off the result from the tableau?

Predicate Logic

When we introduced propositional logic we argued in treating the electrical circuit example, that a string like $high(inv1, o)$ can be read intuitively as ‘the output of inverter 1 is high’, but then we abstracted from this internal structure of the proposition and further on we represented the proposition by a simple string like ‘the_output_of_inverter_1_is_high’. In the formal parts of the calculus we even assumed more abstract just a given countable set of propositions $\{P_1, \dots, P_i, \dots\}$. Until now, we investigated a logical language, which was able to deal with connecting these propositions,

In this section we will have a closer look into the structure of propositions. We will be able to explicitly refer to *objects* like the inverter $inv1$ or its output o in the sentences of our language,. Besides this *domain*, we will introduce the new concept of a *variable*, which stands for an arbitrary object of our domain. Hence, we will be able to write e.g. $high(x, o)$ to denote the fact that the output of a x is high. As a consequence of this

concept we will have *quantifiers*, which allow for sentences like $\exists x \text{ high}(x, o)$, with an intended meaning that there exists an object x , whose output o is high, or $\forall x \text{ high}(x, o)$ with an intended meaning that for all objects x output o is high, Together with these new concepts it will be possible to express rather abstract properties of elements from the domain. Assume, e.g. that we want to introduce the concept of elements in a circuit to be connected. We could say $\text{connected}(\text{inv}1, \text{inv}2)$ and $\text{connected}(\text{inv}2, \text{inv}3)$. Now, with the intended meaning of being connected, we can assume, that $\text{inv}1$ and $\text{inv}3$ are connected as well. Of course, we could introduce the new sentence $\text{connected}(\text{inv}1, \text{inv}3)$ to express this; but the disadvantage is obvious: this have to be done for all objects from our domain. In predicate logic this property of transitivity of connectedness can be expressed without referring to explicit objects by the following sentence:

$$\forall x \forall y \forall z ((\text{connected}(x, y) \wedge \text{connected}(y, z)) \rightarrow \text{connected}(x, z))$$

Syntax

Definition 20 (Syntax of predicate logic – Terms) Assume a

- countable set of function symbols $\{f_i^k \mid i, k = 1, 2, 3, \dots\}$
- a countable set of variables $\{x_i \mid i = 1, 2, 3, \dots\}$

The set of terms is defined by the following induction:

- Variables x_i are terms.
- If t_1, \dots, t_k are terms and f_i^k is a function symbol, then $f_i^k(t_1, \dots, t_k)$ is a term.

Terms of type $f_i^0()$ are special ones, they are called constants. In this case we omit the braces and denote them as f_i^0 .

Terms are the syntactic counterpart of the above mentioned objects. Constants will denote the elements of the domain and function symbols will denote a way to refer to such objects.

The following definition introduces the formulae.

Definition 21 (Syntax of predicate logic – Formulae) Assume a countable set of predicate symbols $\{p_i^k \mid i = 1, 2, 3, \dots\}$.

The set of (well-formed) formulae is defined by the following induction:

- If t_1, \dots, t_k are terms and P_i^k is a predicate symbol, then $P_i^k(t_1, \dots, t_k)$ is a formula.
- If F and G are formulae, then $(F \wedge G)$ and $(F \vee G)$ are formulae.
- If F is a formula, then $\neg F$ is a formula.
- If x is a variable and F a formula, then $\forall x F$ and $\exists x F$ are formulae.

Formulae of type $P_i^k(t_1, \dots, t_k)$ are called atoms or atomic formulae.

Note that the concept of subformulae applies exactly like in the propositional case 5.

We introduce the following abbreviations, which will be used with indices as well:

- $u, v, w, x, z,$ for variables
- a, b, c, \dots for constants
- f, g, h, \dots for function symbols
- p, q, r, \dots for predicate symbols

Note the the arity of function and predicate symbols is omitted in these abbreviations; we assume that it will be obvious from the context.

Example: Assume we want to represent the following equation, which holds for arbitrary elements in a field:

$$x * (y + z) = x * y + x * z$$

The two operators $*$ and $+$ are represented in a predicate logic formula as binary function symbols f_1^2 and f_2^2 , the three variables are x_1, x_2 and x_3 , and the equality relation $=$ is the binary predicate symbol P_1^2 . Altogether we have the following formula in predicate logic:

$$\forall x_1 \forall x_2 \forall x_3 (P_1^2(f_2^2(x_1, f_1^2(x_2, x_3)), f_1^2(f_2^2(x_1, x_2), f_2^2(x_1, x_3))))$$

In the following we will use the obvious and more liberal notation as in the propositional case.

Definition 22 *An occurrence of a variable x in a formula F is called bound, if it occurs in a subformula of F which is of the form $\exists x G$ or $\forall x G$. Otherwise we call the occurrence free.*

A formula, which do not contain a free occurrence of a variable is called closed.

Example: The following formula contains for x and y free and bound occurrences.

$$\forall z(Q(z) \wedge \forall x(P(x, y)) \vee \exists y(P(x, y)))$$

Semantics

Definition 23 (Semantics of predicate logic – Interpretation) *An interpretation is a pair $\mathcal{I} = (U_{\mathcal{I}}, A_{\mathcal{I}})$, where*

- $U_{\mathcal{I}}$ is an arbitrary nonempty set, called domain, or universe.
- $A_{\mathcal{I}}$ is a mapping which associates to
 - a k -ary predicate symbol, a k -ary predicate over $U_{\mathcal{I}}$,
 - a k -ary function symbol, a k -ary function over $U_{\mathcal{I}}$, and
 - a variable an element from the domain.

Let F be a formula and $\mathcal{I} = (U_{\mathcal{I}}, A_{\mathcal{I}})$ be an interpretation. We call \mathcal{I} an interpretation for F , if $A_{\mathcal{I}}$ is defined for every predicate and function symbol, and for every variable, that occurs free in F .

Example: Let $F = \forall x p(x, f(x)) \wedge q(g(a, z))$ and assume the arities of the symbols as written down. In the following we give two interpretations for F :

- $\mathcal{I}_1 = (N_0, A_1)$, such that
 - $A_1(p) = \{(m, n) \mid m, n \in N_0 \text{ and } m < n\}$
 - $A_1(q) = \{n \in N_0 \mid n \text{ is prime}\}$
 - $A_1(f)(n) = n + 1 \forall n \in N_0$
 - $A_1(g)(m, n) = m + n \forall n, m \in N_0$
 - $A_1(a) = 2$
 - $A_1(z) = 3$

Under this interpretation the formula F can be read as “Every natural number is smaller than its successor and the sum of 2 and 3 is a prime number.”

- $\mathcal{I}_2 = (U_2, A_2)$, such that
 - $U_2 = \{a, f(a), g(a, a), f(g(a, a)), g(f(a), f(a)), \dots\}$
 - $A_2(f)(t) = f(t)$ for $t \in U_2$
 - $A_2(g)(t_1, t_2) = g(t_1, t_2)$, if $t_1, t_2 \in U_2$
 - $A_2(a) = a$
 - $A_2(z) = f(f(a))$
 - $A_2(p) = \{p(a, a), p(f(a), f(a)), p(f(f(a)), f(f(a)))\}$
 - $A_2(q) = \{g(t_1, t_2) \mid t_1, t_2 \in U_2\}$

For a given interpretation $\mathcal{I} = (U, A)$ we write in the following $p^{\mathcal{I}}$ instead of $A(p)$; the same abbreviation will be used for the assignments for function symbols and variables.

Definition 24 (Semantics of predicate logic – Evaluation of Formulae) *Let F be a formula and \mathcal{I} an interpretation for F . For terms t which can be composed with symbols from F the value $\mathcal{I}(t)$ is given by*

- $\mathcal{I}(x) = x^{\mathcal{I}}$
- $\mathcal{I}(f(t_1, \dots, t_k)) = f^{\mathcal{I}}(\mathcal{I}(t_1), \dots, \mathcal{I}(t_k))$, if t_1, \dots, t_k are terms and f a k -ary function symbol. (This holds for the case $k = 0$ as well.)

The value $\mathcal{I}(F)$ of a formula F is given by

- $\mathcal{I}(p(t_1, \dots, t_k)) = \begin{cases} \text{true} & \text{if } (\mathcal{I}(t_1), \dots, \mathcal{I}(t_k)) \in p^{\mathcal{I}} \\ \text{false} & \text{otherwise} \end{cases}$
- $\mathcal{I}(F \wedge G) = \begin{cases} \text{true} & \text{if } \mathcal{I}(F) = \text{true} \text{ and } \mathcal{I}(G) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
- $\mathcal{I}((F \vee G)) = \begin{cases} \text{true} & \text{if } \mathcal{I}(F) = \text{true} \text{ or } \mathcal{I}(G) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$

- $\mathcal{I}(\neg F) = \begin{cases} \text{true} & \text{if } \mathcal{I}(F) = \text{false} \\ \text{false} & \text{otherwise} \end{cases}$
- $\mathcal{I}(\forall G) = \begin{cases} \text{true} & \text{if for every } d \in U : \mathcal{I}_{[x/d]}(G) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
- $\mathcal{I}(\exists G) = \begin{cases} \text{true} & \text{if there is a } d \in U : \mathcal{I}_{[x/d]}(G) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$

where, $f_{[x/d]}(y) = \begin{cases} f(y) & \text{if } x \neq y \\ d & \text{otherwise} \end{cases}$

The notions of satisfiable, valid, and \models are defined according to the propositional case 7. Note that, predicate calculus is an extension of propositional calculus: Assume only 0-ary predicate symbols and a formula which contains no variable, i.e. there can be no terms and no quantifier in a well-formed formula.

On the other hand, predicate calculus can be extended: If one allows for quantifications over predicate and function symbols, we arrive at a second order predicate calculus. E.g.

$$\forall p \exists f p(f(x))$$

Another example for a second order formula of is the induction principle from 2.

Problem 1

The interpretation $\mathcal{I} = \text{isgiven}(U_{\mathcal{I}}, A_{\mathcal{I}})$ as follows:

$$\begin{aligned} U_{\mathcal{I}} &= \mathbb{N} \\ p^{\mathcal{I}} &= \{(m, n) \mid m < n\} \\ f^{\mathcal{I}}(m, n) &= m + n \\ x^{\mathcal{I}} = 5 &; y^{\mathcal{I}} = 7 \end{aligned}$$

Determine the value of following terms and formulae:

1. $\mathcal{I}(f(f(x, x), y))$
2. $\mathcal{I}(\forall x \forall y (p(x, y) \vee p(y, x)))$
3. $\mathcal{I}(p(x, x) \rightarrow p(y, x))$
4. $\mathcal{I}(\exists x p(y, x))$

Problem 2

The interpretation $\mathcal{I} = \text{is given}(U_{\mathcal{I}}, A_{\mathcal{I}})$ as follows:

$$\begin{aligned} U_{\mathcal{I}} &= \mathbb{R} \\ P^{\mathcal{I}} &= \{z \mid z \geq 0\} \\ f^{\mathcal{I}}(z) &= z^2 \\ x^{\mathcal{I}} &= \sqrt{2} \\ E^{\mathcal{I}} &= \{(x, y) \mid x = y\} \\ g^{\mathcal{I}}(x, y) &= x + y \\ y^{\mathcal{I}} &= -1 \end{aligned}$$

Determine the value of following terms and formulae:

1. $\mathcal{I}(g(f(x), f(y)))$
2. $\mathcal{I}(\forall x P(f(x)))$
3. $\mathcal{I}(\exists z \forall x \forall y E(g(x, y), z))$
4. $\mathcal{I}(\forall y (E(f(x), y) \rightarrow P(g(x, y))))$

Problem 3

The following formula is given:

$$F = \forall x \forall y \forall z R(h(h(x, y), z), h(x, h(y, z))) \wedge \exists x \exists y \neg R(h(x, y), h(y, x))$$

Indicate a structure \mathcal{A} , which is a model for F and a structure \mathcal{B} which is no model for F !

Equivalence and Normal Forms

Equivalence of formulae is defined as in the propositional case:

Definition 25 *The formulae F and G are called (semantically) equivalent, iff for all interpretations \mathcal{I} for F and G , $\mathcal{I}(F) = \mathcal{I}(G)$. We write $F \equiv G$.*

The equivalences from the propositional case in theorem 5 hold and in addition we have the following cases for quantifiers.

Theorem 14 *The following equivalences hold:*

$$\begin{aligned} \neg \forall x F &\equiv \exists x \neg F \\ \neg \exists x F &\equiv \forall x \neg F \end{aligned}$$

If x does not occur free in G :

$$\begin{aligned} \forall x F \wedge G &\equiv \forall x (F \wedge G) \\ \forall x F \vee G &\equiv \forall x (F \vee G) \\ \exists x F \wedge G &\equiv \exists x (F \wedge G) \\ \exists x F \vee G &\equiv \exists x (F \vee G) \end{aligned}$$

$$\begin{aligned} \forall x F \wedge \forall x G &\equiv \forall x (F \wedge G) \\ \exists x F \vee \exists x G &\equiv \exists x (F \vee G) \\ \forall x \forall y F &\equiv \forall y \forall x F \\ \exists x \exists y F &\equiv \exists y \exists x F \end{aligned}$$

Proof: We will prove only the equivalence

$$\forall x F \wedge G \equiv \forall x (F \wedge G)$$

with x has no free occurrence in G , as an example.

Assume an interpretation $\mathcal{I} = (U, \mathcal{A})$ such that

$$\begin{aligned} &\mathcal{I}(\forall x F \wedge G) = true \\ \text{iff } &\mathcal{I}(\forall x F) = true \text{ and } \mathcal{I}(G) = true \\ \text{iff } &\text{for all } d \in U : \mathcal{I}_{[x/d]}(F) = true \text{ and } \mathcal{I}(G) = true \\ \text{iff } &\text{for all } d \in U : \mathcal{I}_{[x/d]}(F) = true \text{ and } \mathcal{I}_{[x/d]}(G) = true \text{ (} x \text{ does not occur free in } G\text{)} \\ \text{iff } &\text{for all } d \in U : \mathcal{I}_{[x/d]}((F \wedge G)) = true \\ \text{iff } &\mathcal{I}(\forall x (F \wedge G)) = true. \end{aligned}$$

Note that the following symmetric cases do not hold:

$$\forall x F \vee \forall x G \quad \text{is not equivalent to} \quad \forall x (F \vee G)$$

$$\exists x F \wedge \exists x G \quad \text{is not equivalent to} \quad \exists x (F \wedge G)$$

The theorem for substitutivity holds as in the propositional case 4.

Example: Let us transform the following formulae by means of substitutivity and the equivalences from theorem 14:

$$\begin{aligned} &\equiv (\neg(\exists x P(x, y) \vee \forall z Q(z)) \wedge \exists w P(f(a, w))) \\ &\equiv ((\neg\exists x P(x, y) \wedge \neg\forall z Q(z)) \wedge \exists w P(f(a, w))) \\ &\equiv ((\forall x \neg P(x, y) \wedge \exists z \neg Q(z)) \wedge \exists w P(f(a, w))) \\ &\equiv (\exists w P(f(a, w)) \wedge (\forall x \neg P(x, y) \wedge \exists z \neg Q(z))) \\ &\equiv \exists w (P(f(a, w)) \wedge \forall x (\neg P(x, y) \wedge \exists z \neg Q(z))) \\ &\equiv \exists w (\forall x (\exists z \neg Q(z) \wedge \neg P(x, y)) \wedge P(f(a, w))) \\ &\equiv \exists w (\forall x \exists z (\neg Q(z) \wedge \neg P(x, y)) \wedge P(f(a, w))) \\ &\equiv \exists w \forall x \exists z ((\neg Q(z) \wedge \neg P(x, y)) \wedge P(f(a, w))) \end{aligned}$$

Definition 26 Let F be a formula, x a variable and t a term. $F[x/t]$ is obtained from F by substituting every free occurrence of x by t .

Note, that this notion can be iterated: $F[x/t_1][y/t_2]$ and that t_1 may contain free occurrences of y .

Lemma 4 Let F be a formula, x a variable and t a term.

$$\mathcal{I}(F[x/t]) = \mathcal{I}_{[x/\mathcal{I}(t)]}(F)$$

Lemma 5 (Bounded Renaming) Let $F = QxG$ be a formula, where $Q \in \{\forall, \exists\}$ and y a variable without an occurrence in G , then $F \equiv QyG[x/y]$

Definition 27 A formula is called proper if there is no variable which occurs bound and free and after every quantifier there is a distinct variable.

Lemma 6 For every formula F there is a formula G which is proper and equivalent to F .

Proof: Follows immediately by bounded renaming.

Example:

$$F = \forall x \exists x p(x, f(y)) \wedge \forall x (q(x, y) \vee r(x))$$

has the equivalent and proper formula

$$G = \forall x \exists x p(x, f(y)) \wedge \forall z (q(u, y) \vee r(u))$$

Definition 28 A formula is called in prenex form if it has the form $Q_1 \cdots Q_n F$, where $Q_i \in \{\forall, \exists\}$ with no occurrences of a quantifier in F

Theorem 15 For every formula there is a proper formula in prenex form, which is equivalent.

Example:

$$\forall x \exists y p(x, g(y, f(x)) \vee \neg q(z)) \vee \neg \forall x r(x, z)$$

$$\forall x \exists y p(x, g(y, f(x)) \vee \neg q(z)) \vee \exists x \neg r(x, z)$$

$$\forall x \exists y p(x, g(y, f(x)) \vee \neg q(z)) \vee \exists v \neg r(v, z)$$

$$\forall x \exists y \exists v p(x, g(y, f(x)) \vee \neg q(z)) \vee \neg r(v, z)$$

Proof: Induction over the the structure of the formula gives us the theorem for an atomic formula immediately.

- $F = \neg F_0$: There is a $G_0 = Q_1 y_1 \cdots Q_n y_n G'$ with $Q_i \in \{\forall, \exists\}$, which is equivalent to F_0 . Hence we have

$$F \equiv \overline{Q_1} y_1 \cdots \overline{Q_n} y_n \neg G'$$

$$\text{where } \overline{Q_i} = \begin{cases} \exists & \text{if } Q_i = \forall \\ \forall & \text{if } Q_i = \exists \end{cases}$$

- $F = F_1 \circ F_2$ with $\circ \in \{\wedge, \vee\}$. There exists G_1, G_2 which are proper and in prenex form and $G_1 \equiv F_1$ and $G_2 \equiv F_2$. With bounded renaming we can construct

$$G_1 = Q_1 y_1 \cdots Q_k y_k G'_1$$

$$G_2 = Q'_1 z_1 \cdots Q'_l z_l G'_2$$

where $\{y_1, \dots, y_n\} \cap \{z_1, \dots, z_l\} = \emptyset$ and hence

$$F \equiv Q_1 y_1 \cdots Q_k y_k Q'_1 z_1 \cdots Q'_l z_l (G'_1 \circ G'_2)$$

In the following we call proper formulae in prenex form PP-formulae or PPF's.

Definition 29 Let F be a PPF. While F contains a \exists -Quantifier, do the following transformation:

F has the form

$$\forall y_1, \dots, \forall y_n \exists z G$$

where G is a PPF and f is a n -ary function symbol, which does not occur in G .

Let F be

$$\forall y_1, \dots, \forall y_n G[z/f(y_1, \dots, y_n)]$$

If there exists no more \exists -quantifier, F is in Skolem form.

Theorem 16 *Let F be a PPF. F is satisfiable iff the Skolem form of F is satisfiable.*

Proof: Let $F = \forall y_1 \cdots \forall y_n \exists z G$; after one transformation according to the while-loop we have

$$F' = \forall y_1 \cdots \forall y_n \exists z G[z/f(y_1, \cdots, y_n)]$$

where f is a new function symbol.

We have to prove that this transformation is satisfiability preserving:

Assume F' is satisfiable. then there exists a model \mathcal{I} for F' \mathcal{I} is an interpretation for F . From the model property we have for all $u_1, \cdots, u_n \in U_{\mathcal{I}}$

$$\mathcal{I}_{[y_1/u_1] \cdots [y_n/u_n]}(G[z/f(y_1, \cdots, y_n)]) = true$$

From Lemma 4 we conclude

$$\mathcal{I}_{[y_1/u_1] \cdots [y_n/u_n][z/v]}(G) = true$$

where $v = \mathcal{I}(f(u_1, \cdots, u_n))$. Hence we have, that for all $u_1, \cdots, u_n \in U_{\mathcal{I}}$ there is a $v \in U_{\mathcal{I}}$, where

$$\mathcal{I}_{[y_1/u_1] \cdots [y_n/u_n][z/v]}(G) = true$$

and hence we have, that $\mathcal{I}(\forall y_1 \cdots \forall y_n \exists z G) = true$, which means, that \mathcal{I} is a model for F . For the opposite direction of the theorem, assume that F has a model \mathcal{I} . Then we have, that for all $u_1, \cdots, u_n \in U_{\mathcal{I}}$, there is a $v \in U_{\mathcal{I}}$, where

$$\mathcal{I}_{[y_1/u_1] \cdots [y_n/u_n][z/v]}(G) = true$$

Let \mathcal{I}' be an interpretation, which deviates from \mathcal{I} only, by the fact that it is defined for the function symbol f , where \mathcal{I} is not defined. We assume that $f^{\mathcal{I}'}(u_1, \cdots, u_n) = v$, where v is chosen according to the above equation.

Hence we have that for all $u_1, \cdots, u_n \in U_{\mathcal{I}'}$

$$\mathcal{I}'_{[y_1/u_1] \cdots [y_n/u_n][z/f^{\mathcal{I}'}(u_1, \cdots, u_n)]}(G) = true$$

and from Lemma 4 we conclude that for all $u_1, \cdots, u_n \in U_{\mathcal{I}'}$

$$\mathcal{I}'_{[y_1/u_1] \cdots [y_n/u_n]}(G[z/f(y_1, \cdots, y_n)]) = true$$

which means, that $\mathcal{I}'(\forall y_1 \cdots \forall y_n \exists z G[z/f(y_1, \cdots, y_n)]) = true$, and hence \mathcal{I}' is a model for F' .

The above results can be used to transform a Formula into a set of clauses, its clause normal form:

Transformation into Clause Normal Form

Given a first order formula F .

- Transform F into an equivalent proper F_1 by bounded renaming.
- Let y_1, \dots, y_k the free variables from F_1 . Transform F_1 into $F_2 = \exists y_1 \dots \exists y_k F_1$
- Transform F_2 into an equivalent prenex form F_3 .
- Transform F_3 into its Skolemform $F_4 = \forall x_1 \dots \forall x_l G$
- Transform G into its CNF $G' = (\bigwedge_{i=1}^n (\bigvee_{j=1}^m L_{i,j}))$ where $L_{i,j}$ is a literal. This results in $F_5 = \forall x_1 \dots \forall x_l G'$
- Write F_5 as a set of clauses:

$$F_6 = \{C_1, \dots, C_n\}$$

$$\text{where } C_i = \{L_{i,1}, \dots, L_{i,m}\}$$

Interaction: CNF Transformation for Predicate Logic

This is our current set of predicate logic formulas:

Currently empty.

Current formulas manipulation:

Enter some formulas:

Help:

[Delete selected](#) / [Delete all](#) current formulas

[Add to](#) / [Overwrite](#) current formulas

Save/Reload current formulas:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):

[Convert selected](#) / [Convert all](#) formulas to CNF, replacing the current clauses.

And this the resulting clause set (possibly from a previous run):

Currently empty.



Problem 1

Let F be a formula, x a variable and t a term. Then $F[x/t]$ denotes the formula which results from F by replacing every free occurrence of x by t . Give a formal definition of this three argument function $F[x/t]$, by induction over the structure of the formula F .

Problem 2

Show the following semantic equivalences:

1. $\forall x(p(x) \rightarrow (q(x) \wedge r(x))) \equiv \forall x(p(x) \rightarrow q(x)) \wedge \forall x(p(x) \rightarrow r(x))$
2. $\forall x(p(x) \rightarrow (q(x) \vee r(x))) \not\equiv \forall x(p(x) \rightarrow q(x)) \vee \forall x(p(x) \rightarrow r(x))$

Problem 3

Show the following semantic equivalences:

1. $(\forall x p(x)) \rightarrow q(b) \equiv \exists x (p(x) \rightarrow q(b))$
2. $(\forall x r(x)) \vee (\exists y \neg r(y)) \equiv (\forall x r(x)) \rightarrow (\exists y r(y))$

Problem 4

Show that for arbitrary formulae F and G , the following holds:

1. $\forall x(F \vee G) \not\equiv \forall x F \vee \forall x G$
2. If $G = F[x/t]$, then $G \models \exists x F$.

Problem 5

Transform the following formulae in Skolem form !

1. $\forall x \forall y \exists z ((p(x, y) \wedge p(y, z)) \rightarrow \neg p(x, z))$
2. $\forall x (\forall y \exists z p(x, y, z) \wedge \exists z \forall y \neg p(x, y, z))$
3. $(\exists x p(x, y)) \rightarrow (\exists x q(x, x))$

Herbrand Theories

Until now we considered arbitrary interpretations of formulae in predicate logics. In particular we sometimes used numbers as interpretation domain and functions, like addition or successor. In the following, we will concentrate on a special case, the Herbrand interpretation and we will discuss their relation to the general case.

Definition 30 Let S be a set of clauses. The Herbrand universe U_H for S is given by:

- All constants which occur in S are in U_H (if no constant appears in S , we assume a single constant, say a to be in U_H).
- For every n -ary function symbol f in S and every $t_1, \dots, t_n \in U_H$, $f(t_1, \dots, t_n) \in U_H$.

Examples: Given the clause set $S_1 = \{P(a), \neg P(x) \vee P(f(x))\}$, we construct the Herbrand universe

$$U_H = \{a, f(a), f(f(a)), f(f(f(a))), \dots\}$$

For the clause set $S_2 = \{p(x) \vee q(x), r(z)\}$ we get the Herbrand universe $U_H = \{a\}$.

Definition 31 Let S be a set of clauses. An interpretation $\mathcal{I} = (U_{\mathcal{I}}, A_{\mathcal{I}})$ is an Herbrand interpretation iff

- $U_{\mathcal{I}} = U_H$
- For every n -ary function symbol ($n \geq 0$) f and $t_1, \dots, t_n \in U_H$

$$f^{\mathcal{I}}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$$

Note, that there is no restriction on the assignments of relations to predicate symbols (except, that, of course, they have to be relations over the Herbrand universe U_H).

In order to discuss the interpretation of predicate symbols, we need the notion of Herbrand basis.

Definition 32 A ground atom or a ground term is an atom or a term without an occurrence of a variable.

The Herbrand basis for a set of clauses S is the set of ground atoms $p(t_1, \dots, t_n)$, where p is a n -ary predicate symbol from S and $t_1, \dots, t_n \in U_H$

We will notate the assignments of relations to predicate symbols by simply giving a set $I = \{m_1, m_2, \dots, m_n, \dots\}$, where each element is a literal with its atom is from the Herbrand basis.

Examples:

- $S = \{p(x) \vee q(x), r(f(y))\}$
- $U_H = \{a, f(a), f(f(a)), \dots\}$
- $I = \{p(a), q(a), \neg r(a), p(f(a)), q(f(a)), \neg r(f(a)), \dots\}$

Definition 33 Let $\mathcal{I} = (U_{\mathcal{I}}, A_{\mathcal{I}})$ be an interpretation for a set of clauses S ; the Herbrand interpretation \mathcal{I}^* corresponding to \mathcal{I} is a Herbrand interpretation satisfying the following condition:

Let t_1, \dots, t_n be Elements from the Herbrand universe U_H for S . By the interpretation \mathcal{I} every t_i is mapped to a $d_i \in U_{\mathcal{I}}$. If $p^{\mathcal{I}}(d_1, \dots, d_n) = \text{true}(\text{false})$, then $p(t_1, \dots, t_n)$ have to be assigned $\text{true}(\text{false})$ in \mathcal{I}^* .

Let us now state a simple, very obvious lemma, which will help us, to focus on Herbrand interpretation in the following.

Lemma 7 If \mathcal{I} is a model for a set of clauses, then every corresponding Herbrand interpretation \mathcal{I}^* is a model for S

Theorem 17 A set of clauses S is unsatisfiable iff there is no Herbrand model for S .

Proof: If S is unsatisfiable, there is obviously no Herbrand model for S .

Assume that there is no Herbrand model for S and that S is satisfiable. Then, there is a model \mathcal{I} for S and according to lemma 7 the corresponding Herbrand interpretation \mathcal{I}^* is a model for S , which is a contradiction.

Semantic Trees

In this section we introduce the concept of semantic trees, which, then can be used to proof completeness of resolution. For this we assume familiarity with the concepts of trees, binary trees and paths in trees.

Definition 34 A semantic tree for a set of clauses S is a binary tree T with root N_0 , such that the edges are label with literals, build from elements from the Herbrand basis of S , such that:

- If N is an inner node, its two outgoing edges are label with complementary literals A and $\neg A$.
- Every path to a node N in T does not contain complementary literals in $I(N)$, where $I(N)$ is the set of literals which are labels along the edges of the path.

Definition 35 A semantic tree is called complete, if every path contains every atom from the Herbrand basis either positiv or negativ.

Example: $S = \{p(x), q(f(x))\}$ with Herbrand basis

$$\{p(a), q(a), p(f(a)), q(f(a)), p(f(f(a))), \dots\}$$

Note, that for a semantic tree T and a node N the set $I(N)$ can be seen as an assignement of truth-values to ground atoms, as it is done in an Herbrand interpretation. Hence we call $I(N)$ a partial interpretation. A complete semantic tree corresponds to an exhaustive “enumeration” of interpretations.

Definition 36 • A node N of a semantic tree T is a failure node if $I(N)$ falsifies some ground instance of a clause in S , but $I(N')$ does not falsify any ground instance of a clause in S for every ancestor node N' of N .

- A semantic tree T is called closed if every path contains a failure node.
- A node N of a closed semantic tree is called an inference node, if both immediate descendant nodes are failure nodes.

Example: $S = \{p(x), \neg p(x) \vee q(f(x)), \neg q(f(a))\}$ with Herbrand basis

$$\{p(a), q(a), p(f(a)), q(f(a)), p(f(f(a))), \dots\}$$

Let T be a complete semantic tree, than we call T' the correspondig closed tree, if it is obtainable from T by cutting all branches at a failure node.

Theorem 18 (Herbrand’s Theorem – Version 1) A set S of clauses is unsatisfiable, iff for every complete semantic tree for S there is a correspondign finite closed semantic tree.

Proof: Assume S to be unsatisfiable and T a complete semantic tree for S . For every path P we have the set of labels I_B , which is an interpretation, because the tree is complete. Hence, I_B falsifies a ground instance C' of a clause $C \in S$, because S is unsatisfiable. Since there are only finitely many literals in C' , there must be a failure node N_B in a finite distance from the root. Since every path has such a failure node, there is a corresponding closed semantic tree T' , which is finite.

For the opposite direction, assume that for every complete semantic tree T there is a finite closed corresponding tree T' . Then, every path contains a failure node, and hence, every interpretation falsifies S ; hence S is unsatisfiable.

Theorem 19 (Herbrand's Theorem – Version 2) *A set S of clauses is unsatisfiable, iff there is a finite unsatisfiable set S' of ground instances of clauses in S .*

Proof: Assume S to be unsatisfiable and T a complete semantic tree for S . By Herbrand's theorem, version 1, there is a finite closed semantic tree T' for S . Let S' be the set of ground instances of clauses, which are falsified at all failure nodes of T' . S' is finite and is falsified by every interpretation and hence unsatisfiable.

The opposite direction we show by contraposition:

Note, that this version of Herbrand's theorem can be turned directly into a proof procedure:

Given a set of clauses S , for which we want to proof unsatisfiability.

- Generate S'_1, \dots, S'_n, \dots sets of ground instances of clauses of S . Perform a propositional test for unsatisfiability on each of them.
- According to Herbrand's theorem, there is a finite S'_N which is unsatisfiable, if S is unsatisfiable.

1. Problem 1

Assume the following set of clauses:

$$M_0 = \{\{p(x)\}, \{q(x, f(x)), \neg p(x)\}, \{-q(g(y), z)\}\}$$

- (a) Indicate (a) the Herbrand-universe and (b) the Herbrand-basis for M_0 !
- (b) Give a closed semantic tree for M_0 !

2. Problem 2

The following formulae F and G are given.

$$\begin{aligned} F &= \forall y p(y) \wedge \exists z \neg p(z) \\ G &= r(a) \wedge \forall x (r(x) \rightarrow r(f(x))) \end{aligned}$$

Give for F and G (a) a finite and (b) an infinite Herbrand model, or argument if this is not possible.

3. Problem 3

The following sets of clauses are given:

- (a) $M_1 = \{\{p(a, x)\}, \{\neg p(y, b), q(y)\}\}$
- (b) $M_2 = \{\{r(x), r(f(x))\}\}$

Give for each of them (a) a Herbrand universe, (b) Herbrand model and (c) a non-Herbrand model.

Resolution

In the propositional case we defined the resolution inference rule by “cutting away” a pair of complementary literals in two clauses which are resolved upon. In the first order case however this is not always sufficient:

$$\begin{aligned} C_1 &: p(x) \vee q(x) \\ C_2 &: \neg p(f(x)) \end{aligned}$$

In these two clauses there are no complementary literals, however, after substituting the term $f(a)$ for the variable x in C_1 and a for x in C_2 we arrive at:

$$\begin{aligned} C'_1 &: p(f(a)) \vee q(f(a)) \\ C'_2 &: \neg p(f(a)) \end{aligned}$$

Now we can apply the inference rule from propositional logic and arrive at the resolvent $q(f(a))$.

Another possibility is to substitute $f(x')$ for x in C_1 to get

$$C''_1 : p(f(x')) \vee q(f(x'))$$

and then we can have the resolvent $q(f(x'))$ from C''_1 and C_2 , which is in a certain sense more general than the resolvent derived previously.

Definition 37 A substitution σ is a function, which maps variables to terms and which is the identical mapping almost everywhere. Hence it can be represented as

$$\sigma = \{x_1/t_1, \dots, x_n/t_n\}$$

If t_1, \dots, t_n are groundterms, we call σ a ground substitution. The empty substitution is notated by ϵ .

Definition 38 Let $\theta = \{x_1/t_1, \dots, x_n/t_n\}$ be a substitution and E an expression (i.e. a literal or a term), then $E\theta$ is the expression, obtained from E by replacing simultaneously each occurrence of $X_i, 1 \leq i \leq n$ in E by the term t_i .

Example:

With $\theta = \{x/a, y/f(b), z/e\}$ and $E = p(x, y, z)$, we get $E\theta = p(a, f(b), e)$

Definition 39 Let $\sigma = \{x_1/t_1, \dots, x_n/t_n\}$ and $\lambda = \{y_1/s_1, \dots, y_m/s_m\}$ be substitutions. Then the composition of substitutions, denoted by $\sigma \circ \lambda$, is the substitution, which is obtained from $\{x_1/t_1\lambda, \dots, x_n/t_n\lambda, y_1/s_1, \dots, y_m/s_m\}$ by deleting any element $x_j/t_j\lambda$ for which $t_j\lambda = x_j$ and any element y_i/s_i such that $y_i \in \{x_1, \dots, x_n\}$.

Example:

Definition 40 Let $\{E_1, \dots, E_n\}$ be a set of expressions and θ a substitution, θ is unifier for $\{E_1, \dots, E_n\}$ iff

$$E_1\theta = E_2\theta = \dots = E_n\theta$$

A unifier θ is called most general unifier iff for every unifier σ there is a substitution λ such that $\sigma = \theta \circ \lambda$.

In the following we discuss an algorithm for computing most general unifiers. For this we assume a set of terms $\{t_1, \dots, t_n\}$ to be unified. First we transform this into a set of equations by introducing a new variable not yet occurring in this set, say y and by defining the set of equations

$$N = \{y = t_1, \dots, y = t_n\}$$

We will now transform this set such that its unifiers stay invariant, where a σ is a unifier of a set of $\{s_1 = t_1, \dots, s_n = t_n\}$ if $\{s_1\sigma = t_1\sigma, \dots, s_n\sigma = t_n\sigma\}$ holds.

Unification

Given a set of expression. Transform it into a set of equations N as defined above. Apply the following transformation rules as long as possible:

1. $\frac{R \uplus \{t = x\}}{R \uplus \{x = t\}}$ *Orient*
where x is a variable and t a non-variable term
2. $\frac{R \uplus \{s = s\}}{R}$ *Delete*
3. $\frac{R \uplus \{f(s_1, \dots, s_n) = f(t_1, \dots, t_n)\}}{R \uplus \{s_1 = t_1, \dots, s_n = t_n\}}$ *Decompose (Termreduction)*
4. $\frac{R \uplus \{x = t\}}{R[x/t] \uplus \{x = t\}}$ *Eliminate (Elimination of variable I)*
if x not in t , but in R
5. $\frac{R \uplus \{x = y\}}{R[x/y] \uplus \{x = y\}}$ *Coalesce (Elimination of variable II)*
if $x \neq y$ in R
6. $\frac{R \uplus \{f(s_1, \dots, s_m) = g(t_1, \dots, t_n)\}}{\text{FAIL}}$ *Conflict*
if $f \neq g$ or $m \neq n$
7. $\frac{R \uplus \{x = t\}}{\text{FAIL}}$ *Occur Check*
if x in t

Interaction: Unification

This is your set of current terms:

Currently empty.

Current terms manipulation:

Enter some terms:

Help:

[Delete selected](#) / [Delete all](#) current formulas

[Add to](#) / [Overwrite](#) current terms

Save/Reload current term:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):

[Unify selected](#) / [Unify all](#) terms.

This is the result, (possibly from a previous run): Currently none.

Theorem 20 *Let N be a set of expressions. The above unification algorithm terminates. If it returns FAIL, there is no unifier for N otherwise N is transformed into a set of equation $\{y_1 = u_1, \dots, y_m = u_m\}$, which represents the most general unifier for N .*

Definition 41 *Let two or more literals of a clause C have a unifier σ , then $C\sigma$ is called a factor of C .*

Example:

With $C = \{p(x), p(f(y)), \neg q(x)\}$ and $\sigma = \{x/f(y)\}$ we get the factor $C\sigma = \{p(f(y)), \neg q(f(y))\}$

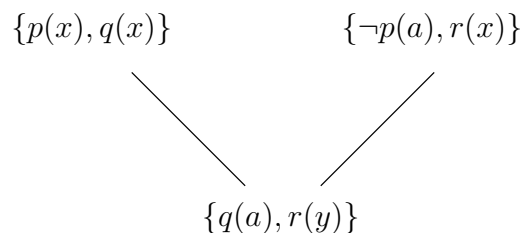
Definition 42 *Let C_1 and C_2 be two clauses with no variables in common, such that $L_1 \in C_1$ and $L_2 \in C_2$ and L_1 and L_2 have a most general unifier σ . A binary resolvent of C_1 and C_2 is*

$$(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$$

Example:

Given $C_1 = \{p(x), q(x)\}$ and $C_2 = \{\neg p(a), r(x)\}$. After renaming C_2 into $C_2 = \{\neg p(a), r(y)\}$ we get the resolvent $\{q(a), r(y)\}$ by using the most general unifier $\{x/a\}$.

We often depict resolvent graphically, e.g.



Definition 43 A resolvent of two clauses C_1 and C_2 is one of the following binary resolvents:

- a binary resolvent of C_1 and C_2
- a binary resolvent of C_1 and a factor of C_2
- a binary resolvent of a factor of C_1 and C_2
- a binary resolvent of a factor of C_1 and a factor of C_2

Example:

Given $C_1 = \{p(x), p(f(y)), r(g(y))\}$ and $C_2 = \{\neg p(f(g(a))), q(b)\}$.

A factor of C_1 is $C_1' = \{p(f(y)), r(g(y))\}$. A binary resolvent of C_1' and C_2 and hence also of C_1 and C_2 is $C_3 = \{r(g(g(a))), q(b)\}$.

Example:

You can use the theorem prover Otter to experiment with resolution:

Interaction: Otter

For informations about Otter look here. First, we need some clauses.

This is our current set of clauses:

Currently empty.

Current clauses manipulation:

Enter some clauses:

Help:

[Delete selected](#) / [Delete all](#) current clauses

[Add to](#) / [Overwrite](#) current clauses

[Convert selected](#) / [Convert all](#) current predicate logic formulas to CNF, replacing the current clauses.

Save/Reload current clauses:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):

[Apply Otter to the current clauses](#) [Delete current Otter Result](#)

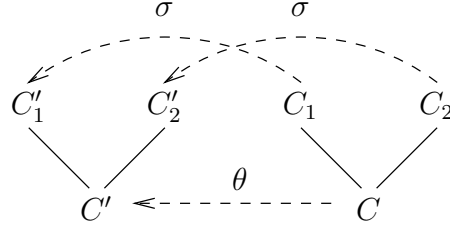
This is the result, (possibly from a previous run):

Currently none.

The following lemma is used in the completeness proof of resolution.



Lemma 8 (Lifting lemma) If C_1' and C_2' are instances of C_1 and C_2 , respectively, and C' is a resolvent of C_1' and C_2' , then there is a resolvent C of C_1 and C_2 such that C' is an instance of C .



Theorem 21 *A set S of clauses is unsatisfiable iff the empty clause can be derived from S by resolution.*

Proof:

Assume that S is unsatisfiable. Let $A = \{A_1, A_2, \dots\}$ be the ground atom set of S , hence the Herbrand basis. Let T be a complete binary tree, as given in Figure ???. According to Herbrand's theorem (version1) 18 there exists a closed finite semantic tree T' . There are two cases:

- If T' consists only of one node (hence the root), The interpretation to be collected from the empty branch in this tree falsifies only the empty clause. Hence the empty clause must be in S .
- Assume T' consists of more than one node. Then there must be an inference node N in T' , hence both its descendants N_1 and N_2 are failure nodes. If such a node would not exist, every node would have at least one non-failure node, which would mean that there is at least an infinite path in T' , which would violate, that fact that it is a finite closed semantic tree.

Let N, N_1, N_2 given as described above; and let

$$\begin{aligned} I(N) &= \{m_1, m_2, \dots, m_n\} \\ I(N_1) &= \{m_1, m_2, \dots, m_n, m_{n+1}\} \\ I(N_2) &= \{m_1, m_2, \dots, m_n, \neg m_{n+1}\} \end{aligned}$$

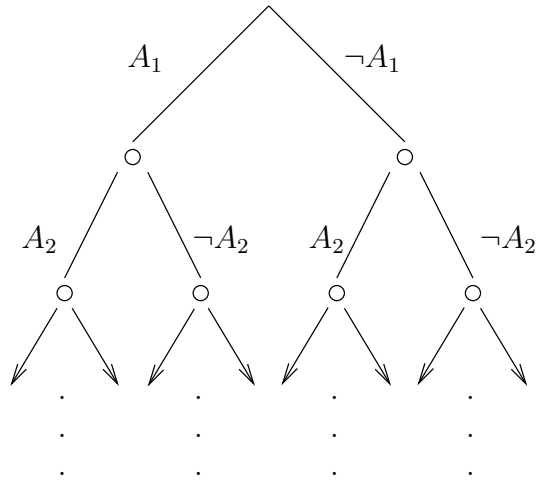
Now, let C'_1 and C'_2 be ground instances of clauses C_1 and C_2 , such that C'_1 is falsified by $I(N_1)$ and C'_2 by $I(N_2)$, such that both are not falsified by $I(N)$. Hence we have $\neg m_{n+1} \in C'_1$ and $m_{n+1} \in C'_2$ and we can construct the resolvent

$$C' = (C'_1 - \{\neg m_{n+1}\}) \cup (C'_2 - \{m_{n+1}\})$$

C' must be false in $I(N)$, because both $(C'_1 - \neg m_{n+1})$ and $(C'_2 - m_{n+1})$ are false in $I(N)$. According to the Lifting Lemma 8 there exists a resolvent C of C_1 and C_2 , such that C' is a ground instance of C . Let T'' be the closed semantic tree for $S \cup \{C\}$, obtained from T' by deleting all nodes below the first node which falsifies C' . Note, that S is unsatisfiable if and only if $S \cup \{C\}$ is unsatisfiable. Clearly, T'' has less nodes than T' and we now can iterate this process until only the root of the semantic tree is remaining. This, however is only possible if the empty clause \square is derivable.

For the opposite direction, assume that \square is derivable by resolution from S and let R_1, \dots, R_k the resolvents constructed during this process. Assume S is satisfiable and M to be a model for S . From the correctness lemma according to the propositional case we know, that if a model satisfies two clauses it also satisfies its resolvent. Therefore M has to satisfy R_1, \dots, R_k ; this, however, is impossible, because one of this resolvents is \square .

T :



Problem 1

Indicate in each case a derivation of the empty clause with predicate-logical resolution!

1. $\{\{p(x, 0, x)\}, \{p(x, s(y), s(z)), \neg p(x, y, z)\}, \{\neg p(s(s(s(0))), s(s(0)), u)\}\}$
2. $\{\{q(x), q(s(x))\}, \{\neg q(x), \neg q(s(s(x)))\}\}$ (★)
3. $\{\{\neg r(x, f(x), y), \neg r(x, g(y), z)\}, \{r(c, u, i(v)), r(h(u), v, j(v))\}\}$

Problem 2

Show the following *Lifting lemma* by means of induction over the term- and formula construction:

Is F a predicate-logical formula, and \mathcal{I} a fitting interpretation for F and $F[x/t]$. Then

$$\mathcal{I}(F[x/t]) = \mathcal{I}_{[x/\mathcal{I}(t)]}(F),$$

is valid, if t does not contain any variable that $[x/t]$ is laced by the substitution in F .

Problem 3

Compute - if possible - the most general unifier of following sets of clauses:

1. $\{p(x, a), p(f(c), y)\}$
2. $\{p(f(x), a, x), p(y, z, z)\}$
3. $\{q(x, x), q(g(y), y)\}$
4. $\{r(x, x), r(a, h(y))\}$

Problem 4

Determine all direct resolvents of the following pairs of clauses:

1. $\{\neg p(x), q(x, b)\}$ und $\{p(a), q(a, b)\}$
2. $\{p(x), p(f(x))\}$ und $\{\neg p(x), \neg p(f(f(x)))\}$
3. $\{\neg q(c, g(c))\}$ und $\{\neg p(x), q(x, x)\}$
4. $\{\neg p(x, y, z), \neg p(y, u, v), \neg p(x, v, w), p(z, u, w)\}$ und $\{p(g(x, y), x, y)\}$

Problem 5

Compute - if possible - the most general unifier of following sets of clauses:

1. $\{o(x, x), o(a, f(y))\}$
2. $\{p(x, a), p(f(c), y)\}$
3. $\{q(g(x), a, x), q(y, z, z)\}$
4. $\{r(x, x), r(h(y), y)\}$

Problem 6

Determine all direct resolvents of the following pairs of clauses:

1. $\{\neg p(x), \neg p(b), q(x, b)\}$ and $\{p(a), q(a, b)\}$
2. $\{r(x), r(f(x))\}$ and $\{\neg r(x), \neg r(f(f(x)))\}$
3. $\{\neg s(c, g(c))\}$ and $\{s(x, x), \neg t(x)\}$

Problem 7

Give for the following set of clauses (a) a linear derivation, (b) a derivation with unit resolution, (c) a further (maximally short) derivation of the empty clause by means of predicate-logical resolution!

$$\{\{\neg e(x), o(s(x))\}, \{\neg e(x), \neg o(s(x)), e(s(s(x)))\}, \{e(a)\}, \{\neg o(s(s(s(s(s(a))))))\}\}$$

Problem 8

Indicate in each case a derivation of the empty clause with predicate-logical resolution!

1. $\{\{p(x, 0, x)\}, \{p(x, s(y), s(z)), \neg p(x, y, z)\}, \{\neg p(s(s(s(0))), s(s(0)), u)\}\}$
2. $\{\{q(x), q(s(x))\}, \{\neg q(x), \neg q(s(s(x)))\}\}$ (★)
3. $\{\{\neg r(x, f(x), y), \neg r(x, g(y), z)\}, \{r(c, u, i(v)), r(h(u), v, j(v))\}\}$

Strategies for Resolution

Linear Resolution In contrast to the saturation-based procedure, which we gave for propositional resolution, we will discuss now a strategy which allows goal directed generation of resolvents. We will see later, namely in the case of Horn clauses, that this linear strategy, is the basis for the interpretation of logic programs.

Definition 44 Given a set of clauses S and a clause C_0 in S . A linear deduction of top clause C_0 is a sequence C_0, C_1, \dots, C_n , where $\forall 1 \leq i \leq n$ C_i is a resolvent of C_i and B with $B \in S \cup \{C_j \mid j < i\}$.

If $C_n = \square$ the sequence is called a linear refutation.

The following is an example for a linear deduction. The clause set S is given by:

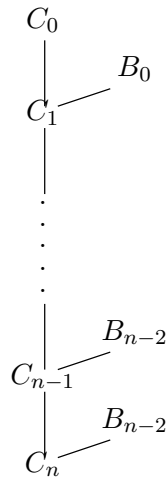
- $$\begin{aligned}
 & \text{symptom}(s) \quad (1) \\
 & \text{cause}(c_1) \vee \text{cause}(c_2) \vee \neg \text{symptom}(s) \quad (2) \\
 & \text{treatment}(t_0) \vee \neg \text{cause}(c_1) \quad (3) \\
 & \text{treatment}(t_1) \vee \neg \text{cause}(c_1) \quad (4) \\
 & \text{treatment}(t_0) \vee \neg \text{cause}(c_2) \quad (5) \\
 & \text{treatment}(t_2) \vee \neg \text{cause}(c_2) \quad (6)
 \end{aligned}$$

and together with the goal $\neg \text{treatment}(x)$, we get the following refutation, where clauses from S are given by the respective numbers:

$\neg \text{treatment}(x)$, (4), $\neg \text{cause}(c_1)$, (2), $\text{cause}(c_2) \vee \neg \text{symptom}(s)$, (1), $\text{cause}(c_2)$, (6), $\text{treatment}(t_2)$, \square

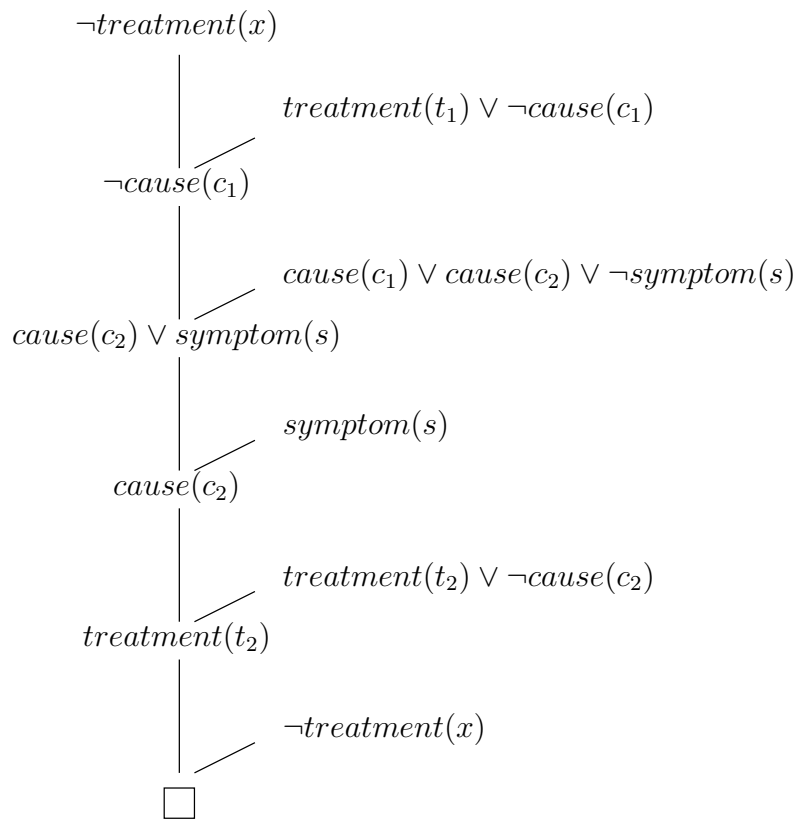
the same refutation can be given more naturally by the following picture:

The following theorem states correctness and completeness of linear resolution. Note that completeness only states that there exists a linear refutation, there is no guaranty that every clause in the sequence really is necessary to derive the empty clause.



Theorem 22 *Linear resolution is complete and correct.*

Example



Input and Unit Resolution

Definition 45 Given a set of clauses S . The inference rule input resolution is resolution, such that one parent clause is a clause from S .

The inference rule unit resolution is resolution, such that at least one parent clause is a unit clause or a unit factor of a parent clause.

In an obvious way the notions of unit (input) derivations and refutations are defined.

Theorem 23 For a set S of clauses there exists a unit refutation iff there exists an input refutation.

Unit resolution (and hence input resolution) is not complete for full first order logics! For Horn clauses, however, it is a complete strategy and indeed, it is the basis for the SLD-resolution principle, which is the core of

SLD-Resolution In this section we will introduce a special form of linear resolution for Horn clauses. We will interpret a clause a program by notating it in the following form:

A	\leftarrow		program clause (fact)
A	\leftarrow	B_1, \dots, B_n	program clause
	\leftarrow	B_1, \dots, B_n	goal

Definition 46 Let P be a set of program clauses. Assume a selection function, which gives for a given goal $\leftarrow A_1, \dots, A_n$ one of its subgoals A_i .

Further assume a goal $G_i = \leftarrow A_1, \dots, A_m, \dots, A_n$ and a selection function which selects A_m . Let $C_i = A \leftarrow B_1, \dots, B_q$ be a variant of a clause in P , such that C_i and G_i have no variable in common. If θ_{i+1} is most general unifier of A_m and A , the goal

$$G_{i+1} = \leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_n)\theta_{i+1}$$

is called SLD-resolvent

Definition 47 An SLD-deduction (-refutation) of $P \cup \{G\}$ for a set of program clauses P and a goal clause G is a linear deduction (refutation) in which only SLD-resolution steps occur and G is the start clause.

- An R -computed answer substitution θ for $P \cup \{G\}$ is $\theta_1 \circ \dots \circ \theta_n|_{\text{Var}(G)}$, where $\theta_1, \dots, \theta_n$ are the mgUs from a SLD-refutation of $P \cup \{G\}$ with selection function R .
- A substitution θ for $\text{Var}(G)$ is an answer substitution for $P \cup \{G\}$.
- It is a correct answer substitution for $P \cup \{G\}$, if $P \models \forall G\theta$

Theorem 24 Let P be a set of program clauses, G a goal clause and R a selection function. For every correct answer substitution θ for $P \cup \{G\}$, there is an R -computed answer substitution σ for $P \cup \{G\}$ and a substitution γ , such that $\theta = \sigma \circ \gamma|_{\text{Var}(G)}$

Model Elimination

In the Section on Propositional Logic we already explained propositional tableaux and its variants, like the connection calculus and model elimination. In this section we will give model elimination in the first order case. Note that we need one more inference rule, the reduction rule, in this case

Definition 48 A clause (normalform) tableau for a set of clauses S is a tableau for S , whose nodes are literals from S and which is constructed by a (possibly infinite) sequence of applications of the following rules:

- The tree consisting of root true and immediate successors L_1, \dots, L_n , where $C = L_1, \dots, L_n$ is a new variant of a clause from S is a tableau for S (initialisation rule).
- Let T be a tableau for S , B a branch of T , and $C = L_1, \dots, L_n$ an new variant of a clause from S , such that the link-condition with mgu σ is satisfied. If the tree T' is constructed by extending B by the n subtrees L_i , then $T'\sigma$ is a tableau for S (expansion rule).
- Let T be a tableau for S , B a branch of T , L a leaf of B , and $L' \in C$, such that $\overline{L'}$ and L have a mgu σ , than $T\sigma$ is a tableau for S (reduction rule).

The following are three possible link conditions:

1. No condition.
2. *Weak link condition*: There is a literal $L \in B$ and $L' \in C$, such that $\overline{L'}$ and L have a mgu σ
3. *Strong link condition*: There is a leaf L of B , and $L' \in C$, such that $\overline{L'}$ and L have a mgu σ .

Analog to the propositional case the different link conditions result in different calculi:

- The empty condition results in a clause normal form tableau calculus.
- The weak condition results in a *connection calculus*.
- The strong link condition results in a *model elimination calculus*.

A Prolog-like Implementation

```
% gprove(G,A) is true if G <- A follows from KB
gprove(true,_).
gprove((G & H),A):-
    gprove(G,A),
    gprove(H,A).
gprove(G,A):-
    member(G,A).
gprove(G,A):-
    (G <- B),
    neg(G,NG),
    gprove(B,[NG|A]).

% neg denotes the negation of atoms

% to prove a theorem do
% gprove(yes, []).

% (a) Set of contrapositives

p(a,X) <- ~p(b,Y) & q(X,Y).
p(b,Y) <- ~p(a,X) & q(X,Y).
~q(X,Y) <- ~p(a,X) & ~p(b,Y).
p(Z,Z) <- true.
q(b,a) <- true.
yes <- p(U,V).
~p(U,V) <- ~yes.
```

Iterativ deepening Iterative deepening is a complete search strategy, which combines depth-first with breadth-first search.

- $depthbound = 1$
- do while (not found)
 - Search by limited depth-first search until $depthbound$.
 - Set $depthbound = depthbound + 1$

Number of expansions in a tree with branching factor b until depth d is

$$1 + b + b^2 + \dots + b^{d-1} + b^d$$

The total number of expansions in iterative deepening search is

$$(d + 1) * 1 + d * b + (d - 1) * b^2 + \dots + 3 * b^{d-2} + 2 * b^{d-1} + 1 * b^d$$

Hence time complexity of iterative deepening is still $O(b^d)$.

PTTP- Prolog Technology Theorem Proving

As exemplified by PTTP (“Prolog Technology Theorem Prover”) [?, ?], Prolog can be viewed as an “almost complete” theorem prover, which has to be extended by only a few ingredients in order to handle the non-Horn case. By this technique the benefits of optimizing Prolog compilers are accessible to theorem proving. First we will briefly review the standard approach, and then we will describe the necessary modifications to obtain restart model elimination.

The PTTP-approach transforms a given clause set into a Prolog program. The transformed Prolog program must execute the clauses according to some complete proof procedure. *Model elimination* turns out to be particularly useful for this, since it is, like Prolog, an input proof procedure. In particular, the transformation from the input clauses to Prolog works as follows:

- An input clause such as

$$C \leftarrow A \wedge B$$

is transformed into a Prolog clause

$$(1) \quad c :- a, b.$$

Additionally, since in the model elimination calculus every literal in a clause can equally well serve as an entry point into the clause, all contrapositives are needed. In this case these are

$$(2) \quad \text{not_a} :- \text{not_c}, b.$$

$$(3) \quad \text{not_b} :- a, \text{not_c}.$$

This example also shows how negation is treated, namely by making it part of the predicate name.

- Prolog’s unsound unification has to be replaced by a sound unification algorithm. This can either be done by directly building-in sound unification into the Prolog implementation, or by reprogramming sound unification in Prolog and calling this code instead of Prolog’s unsound unification.

- A complete search strategy is needed. Usually depth bounded iterative deepening is used. The strategy can be compiled into the prolog program by additional parameters, being used as “current depth” and “limit depth”. The cost of an extension step can be uniformly 1 (depth bounded search), or can be proportional to the length of the input clause (inference bounded search).
- The model elimination reduction operation has to be implemented. This can be realized by memorizing the subgoals solved so far (the A-literals) as a list in an additional argument, and by Prolog code that checks a goal for a complementary member of that list. Of course, this check has to be carried out with sound unification.

The Prolog clause (1) from above then looks like

(1') $c(\text{Anc}) \text{ :- } a([-a|\text{Anc}]), b([-b|\text{Anc}]).$

where `Anc` is a Prolog list which contains the ancestor literals (called A-literals in [?]); code for reduction steps then looks like

(Red-C) $c(\text{Anc}) \text{ :- } \text{member}(c, \text{Anc}).$

(Red-notC) $\text{not_c}(\text{Anc}) \text{ :- } \text{member}(-c, \text{Anc}).$

Model Generation Theorem Proving

SATCHMO

The SATCHMO Theorem Prover was one of the first systems which used model generation, i.e. a bottom-up proof procedure. The prover was given by a small Prolog-program, which implements a tableau proof procedure. One restriction is that it requires range restricted formulae.

Definition 49 *A first order clause $A_1 \vee \dots \vee A_n \leftarrow B_1 \wedge \dots \wedge B_m$ is called range restricted if every variable which occurs in the head $A_1 \vee \dots \vee A_n$ occurs in the body $B_1 \wedge \dots \wedge B_m$ as well.*

1. Convert clauses to range restricted form:

$$q(x) \vee p(x, y) \leftarrow q(x) \quad \rightsquigarrow \quad q(X) ; p(X, Y) \leftarrow q(X), \text{dom}(Y)$$

2. `assert` range-restricted clauses and `dom` clauses in Prolog database.
3. Call `satisfiable`:

```

satisfiable :-
    (Head <- Body),
    Body, not Head, !,
    component(HLit, Head),
    assume(HLit),
    not false,
    satisfiable.
satisfiable.

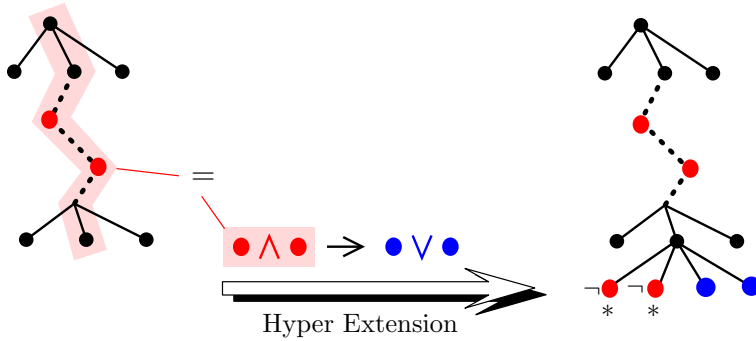
assume(X) :- asserta(X).
assume(X) :-
    retract(X), !, fail.
component(E, (E ; _)).
component(E, (_ ; R)) :-
    !, component(E, R).
component(E, E).

```

First-Order completeness via Level-Saturation modification.

This proof procedure implements Hyper Tableaux in the ground case.

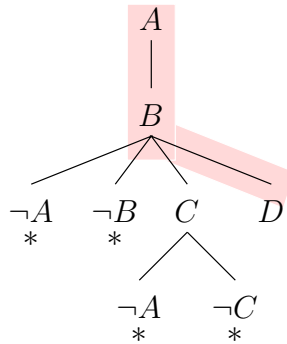
Hyper Tableau - Ground Case



All open branches consist of positive literals only

Take the following clause set as an example

$\{\rightarrow A, \rightarrow B, A \wedge B \rightarrow C \vee D, A \wedge B \rightarrow E \vee D, A \wedge C \rightarrow\}$



Definition 50 (Literal tree, Clausal Tableau [?]) A literal tree is a pair (t, λ) consisting of a finite, ordered tree t and a labeling function λ that assigns a literal to every non-root node of t . The successor sequence of a node N in an ordered tree t is the sequence of nodes with immediate predecessor N , in the order given by t .

A (clausal) tableau T of a set of clauses \mathcal{S} is a literal tree (t, λ) in which, for every successor sequence N_1, \dots, N_n in t labeled with literals K_1, \dots, K_n , respectively, there is a substitution σ and a clause $\{L_1, \dots, L_n\} \in \mathcal{S}$ with $K_i = L_i\sigma$ for every $1 \leq i \leq n$. $\{K_1, \dots, K_n\}$ is called a tableau clause and the elements of a tableau clause are called tableau literals.

Definition 51 (Branch, Open and Closed Tableau, Selection Function) A branch of a tableau T is a sequence N_0, \dots, N_n ($n \geq 0$) of nodes in T such that N_0 is the root of T , N_i is the immediate predecessor of N_{i+1} for $0 \leq i < n$, and N_n is a leaf of T . We say branch $b = N_0, \dots, N_n$ is a prefix of branch c , written as $b \leq c$ or $c \geq b$, iff $c = N_0, \dots, N_n, N_{n+1}, \dots, N_{n+k}$ for some nodes N_{n+1}, \dots, N_{n+k} , $k \geq 0$.

The branch literals of branch $b = N_0, \dots, N_n$ are the set $\text{lit}(b) = \{\lambda(N_1), \dots, \lambda(N_n)\}$. We find it convenient to use a branch in place where a literal set is required, and mean its branch literals. For instance, we will write expressions like $A \in b$ instead of $A \in \text{lit}(b)$.

In order to memorize the fact that a branch contains a contradiction, we allow to label a branch as either open or closed. A tableau is closed if each of its branches is closed, otherwise it is open.

A selection function is a total function f which maps an open tableau to one of its open branches. If $f(T) = b$ we also say that b is selected in T by f .

Note that branches are always finite, as tableaux are finite.

Fortunately, there is no restriction on which selection function to use. For instance, one can use a selection function which always selects the “leftmost” branch.

Definition 52 (Hyper Tableau - Ground Case) Let S be a finite set of clauses and f be a selection function. Hyper tableaux for S are inductively defined as follows:

Initialization step: A one node literal tree is a hyper tableau for S . Its single branch is marked as “open”.

Hyper extension step: If

1. T is an open hyper tableau for S , $f(T) = b$ (i.e. b is selected in T by f) with open leaf node N , and
2. $C = A_1, \dots, A_m \leftarrow B_1, \dots, B_n$ is a clause from S ($m \geq 0, n \geq 0$), called extending clause in this context, and
3. such that $\{B_1, \dots, B_n\} \subseteq b$ (referred to as hyper condition)

then the literal tree T' is a hyper tableau for S , where T' is obtained from T by attaching $m + n$ child nodes $M_1, \dots, M_m, N_1, \dots, N_n$ to b with respective labels

$$A_1, \dots, A_m, \neg B_1, \dots, \neg B_n$$

and marking every new branch $(b, M_1), \dots, (b, M_m)$ with positive leaf as “open”, and marking every new branch $(b, N_1), \dots, (b, N_n)$ with negative leaf as “closed”.

Minimal Model Reasoning

The clause set $M = \{A \vee B \leftarrow, B \leftarrow A\}$ obviously has two different models: $\{A, B\}$ and $\{B\}$. Under set inclusion these models can be compared and there are some tasks where it is appropriate to compute the (or in general a) smallest one. This is for example the case with

- Knowledge Representation, Circumscription
- Basis for default negation (GCWA)
- Applications: Deductive database updates, Diagnosis

There are basically two different methods to compute minimal models.

Minimal Model Reasoning – Niemelä’s Approach

Given a set of ground clauses M the method applies a model generating procedure, e.g. hyper tableau, which is able to generate all models.

Lemma 1: For every minimal model p for M there is a branch with literals p .

Assume that Σ is the set of atoms, which occur in the head of a clause from M , then the following Lemma holds.

Lemma 2: p is a minimal model for M iff $M \cup \{\neg A \mid A \in \Sigma \setminus p\} \models p$

This offers a general method: Generate model candidates, and test with Lemma 2.

$p = \{A, B\}$ is not a minimal model in our example from above, because $M \cup \{p\} \models \{A, B\}$ iff $M \cup \{\leftarrow A \wedge B\}$ is unsatisfiable, which is not the case, hence p does not correspond to a minimal model and hence the branch is closed.

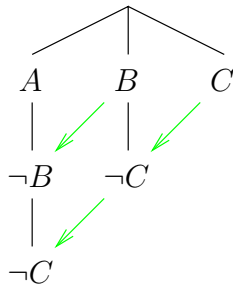
$p = \{B\}$ is minimal because

$M \cup \{\leftarrow A\} \models \{B\}$ iff $M \cup \{\leftarrow A\} \cup \{\leftarrow B\}$ is unsatisfiable. This is the case and hence p is minimal and the branch remains open.

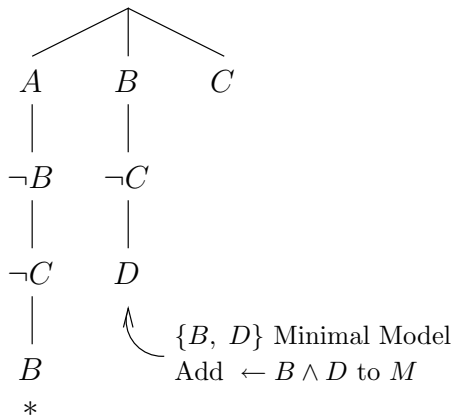
Properties: Soundness (by Lemma 2) Completeness (by Lemma 1), *space efficiency*.

Minimal Model Reasoning – Bry& Yayha’s Approach

As an example we have the set $M = \{A \vee B \vee C \leftarrow, B \leftarrow A, D \leftarrow B\}$



Complement Splitting



Lemma: With complement splitting, the leftmost open branch is a minimal model for M .

General method: Repeat: generate minimal model p , add $\leftarrow p$ to M .

Properties: Soundness (by Lemma) Completeness as before,

possibly exponentially many new clauses $\leftarrow p$.

In the Section on Propositional Logic we already explained propositional tableaux and its variants, like the connection calculus and model elimination. In this section we will give model elimination in the first order case. Note that we need one more inference rule, the reduction rule, in this case

Definition 53 A clause (normalform) tableau for a set of clauses S is a tableau for S , whose nodes are literals from S and which is constructed by a (possibly infinite) sequence of applications of the following rules:

- The tree consisting of root true and immediate successors L_1, \dots, L_n , where $C = L_1, \dots, L_n$ is a new variant of a clause from S is a tableau for S (initialisation rule).
- Let T be a tableau for S , B a branch of T , and $C = L_1, \dots, L_n$ an new variant of a clause from S , such that the link-condition with mgu σ is satisfied. If the tree T' is constructed by extending B by the n subtrees L_i , then $T'\sigma$ is a tableau for S (expansion rule).
- Let T be a tableau for S , B a branch of T , L a leaf of B , and $L' \in C$, such that $\overline{L'}$ and L have a mgu σ , than $T\sigma$ is a tableau for S (reduction rule).

The following are three possible link conditions:

1. No condition.
2. *Weak link condition*: There is a literal $L \in B$ and $L' \in C$, such that $\overline{L'}$ and L have a mgu σ
3. *Strong link condition*: There is a leaf L of B , and $L' \in C$, such that $\overline{L'}$ and L have a mgu σ .

Analog to the propositional case the different link conditions result in different calculi:

- The empty condition results in a clause normal form tableau calculus.
- The weak condition results in a *connection calculus*.
- The strong link condition results in a *model elimination calculus*.

Protein

This is our current set of clauses:

Currently empty.

Current clauses manipulation:

Enter some clauses:

Help:

[Delete selected](#) / [Delete all](#) current clauses

[Add to](#) / [Overwrite](#) current clauses

[Convert selected](#) / [Convert all](#) current predicate logic formulas to CNF, replacing the current clauses.

Save/Reload current clauses:

[Save](#) to database (enter name):

[Reload](#) from database (select name):

[Reload](#) from file (enter file name):



Apply Protein to the current clauses Delete current Protein result

This is the result, (possibly from a previous run):

Currently none.

A Prolog-like Implementation

```
% gprove(G,A) is true if G <- A follows from KB
gprove(true,_).
gprove((G & H),A):-
    gprove(G,A),
    gprove(H,A).
gprove(G,A):-
    member(G,A).
gprove(G,A):-
    (G <- B),
    neg(G,NG),
    gprove(B,[NG|A]).
```

```
% neg denotes the negation of atoms
```

```
% to prove a theorem do
% gprove(yes, []).
```

```
% (a) Set of contrapositives
```

```
p(a,X) <- ~p(b,Y) & q(X,Y).
p(b,Y) <- ~p(a,X) & q(X,Y).
~q(X,Y) <- ~p(a,X) & ~p(b,Y).
p(Z,Z) <- true.
q(b,a) <- true.
yes <- p(U,V).
~p(U,V) <- ~yes.
```

Iterative deepening Iterative deepening is a complete search strategy, which combines depth-first with breadth-first search.

- $depthbound = 1$
- do while (not found)
Search by limited depth-first search until $depthbound$.
Set $depthbound = depthbound + 1$

Number of expansions in a tree with branching factor b until depth d is

$$1 + b + b^2 + \dots + b^{d-1} + b^d$$

The total number of expansions in iterative deepening search is

$$(d+1) * 1 + d * b + (d-1) * b^2 + \dots + 3 * b^{d-2} + 2 * b^{d-1} + 1 * b^d$$

Hence time complexity of iterative deepening is still $O(b^d)$.

1 PTTP- Prolog Technology Theorem Proving

As exemplified by PTTP (“Prolog Technology Theorem Prover”) [?, ?], Prolog can be viewed as an “almost complete” theorem prover, which has to be extended by only a few ingredients in order to handle the non-Horn case. By this technique the benefits of optimizing Prolog compilers are accessible to theorem proving. First we will briefly review the standard approach, and then we will describe the necessary modifications to obtain restart model elimination.

The PTTP-approach transforms a given clause set into a Prolog program. The transformed Prolog program must execute the clauses according to some complete proof procedure. *Model elimination* turns out to be particularly useful for this, since it is, like Prolog, an input proof procedure. In particular, the transformation from the input clauses to Prolog works as follows:

- An input clause such as

$$C \leftarrow A \wedge B$$

is transformed into a Prolog clause

$$(1) \quad c :- a, b.$$

Additionally, since in the model elimination calculus every literal in a clause can equally well serve as an entry point into the clause, all contrapositives are needed. In this case these are

$$(2) \quad \text{not_a} :- \text{not_c}, b.$$

$$(3) \quad \text{not_b} :- a, \text{not_c}.$$

This example also shows how negation is treated, namely by making it part of the predicate name.

- Prolog’s unsound unification has to be replaced by a sound unification algorithm. This can either be done by directly building-in sound unification into the Prolog implementation, or by reprogramming sound unification in Prolog and calling this code instead of Prolog’s unsound unification.
- A complete search strategy is needed. Usually depth bounded iterative deepening is used. The strategy can be compiled into the prolog program by additional parameters, being used as “current depth” and “limit depth”. The cost of an extension step can be uniformly 1 (depth bounded search), or can be proportional to the length of the input clause (inference bounded search).
- The model elimination reduction operation has to be implemented. This can be realized by memorizing the subgoals solved so far (the A-literals) as a list in an additional argument, and by Prolog code that checks a goal for a complementary member of that list. Of course, this check has to be carried out with sound unification.

The Prolog clause (1) from above then looks like

$$(1') \quad c(\text{Anc}) :- a([-a|\text{Anc}]), b([-b|\text{Anc}]).$$

where `Anc` is a Prolog list which contains the ancestor literals (called A-literals in [?]); code for reduction steps then looks like

```
(Red-C)      c(Anc) :- member(c,Anc).
(Red-notC)   not_c(Anc) :- member(-c,Anc).
```

Modal Logic

Modal logic is concerned with the investigation of modalities in logics. Modalities are e.g. *necessity*, *possibility*, *provability* or *uncertainty*. In classical propositional logics, propositions like *married(tom,mary)* are *true* or *false*; in real life however, the truthvalue of the above proposition obviously can change in time. In a different context the truthvalue may be different in different worlds: assume that Tom is dreaming about being married with Mary; hence in the world of his dreams the proposition may be true, while in real-life it may be false. In still another context, the truthvalue can depend whether it is considered under legal aspects: it is possible that Tom and Mary are legally married, while the catholic church is considering them to be singles.

Modal logics have been studied extensively already during the first half of the 20th century by various logicians. The main breakthrough, however, was the establishment of a formal semantics of modal logic given by Kripke.

Propositional logics can be extended by modalities to describe *believe*, *knowledge* or *temporal aspects*. Hence it is very appropriate to be used with knowledge representation systems. Recently modal logics are applied in verification contexts and as a means to describe the semantics of description logics.

As an example take the following puzzle:

Assume 3 children (works for n as well), who are perfect reasoners, are always truthful, and always given an answer, if they know one. The children are playing outside, and may get muddy foreheads. Any child can see if the other children have muddy foreheads, but can't see his or her own forehead. At any point, an adult says to them: at least one of you has mud on your forehead.

The adult now says: Do any of you know if you have mud on your forehead. No one answers. The adult repeats the question, and again no one answers. The 3rd time the adult asks the question, one or more of the children answer.

How many of the children have muddy foreheads?

This puzzle can be solved by constructing a Kripke structure, which is a set of states together with links which express the accessibility of states. There are 3 children, each can be muddy or not muddy and, hence, we have 2^3 possible states. States can be represented by triple of booleans, where a 1 in the n th position means that the n th child is muddy, and a 0 in the n th position that the n th child is not muddy.

The Kripke structure can be defined as follows: consider, e.g. state (111). Since child 1 does not know whether or not he or she is muddy, as far as child 1 is concerned, he or she could be in state (011). For child 2 the state (011), however, is not accessible, since child 2 knows that child 1 is muddy.

A structure which is constructed as depicted above, can be used to solve the puzzle, by drawing the structure as it results after each speech action of the adult.

Syntax

We assume the syntax of classical propositional logic as used in the chapters above. Additionally we have the following two rules:

If A is a formula

$$\begin{array}{l} \diamond A \text{ and} \\ \square A \end{array}$$

are formulae.

The symbols \diamond and \square stand traditionally for *possibility* and *necessity*; in the context of temporal logic they stand for *always* and *eventually*, so that $\diamond A$ stands for *A is eventually true* and $\square A$ for *A is always true*.

Kripke Semantics

Definition 54 A Kripke frame is a pair $\langle W, R \rangle$, where W is a non-empty set (of possible worlds) and R a binary relation on W . We write wRw' iff $(w, w') \in R$ and we say that world w' is accessible from world w , or that w' is reachable from w , or that w' is a successor of w .

A Kripke model is a triple $\langle W, R, V \rangle$, with W and R as above and V is a mapping $\mathcal{P} \mapsto 2^W$, where \mathcal{P} is the set of propositional variables. $V(p)$ is intended to be the set of worlds at which p is true under the valuation V .

Given a model $\langle W, R, V \rangle$ and a world $w \in W$, we define the satisfaction relation \models by:

$$\begin{array}{ll} w \models p & \text{iff } w \in V(p) \\ w \models \neg A & \text{iff } w \not\models A \\ w \models A \wedge B & \text{iff } w \models A \text{ and } w \models B \\ w \models A \vee B & \text{iff } w \models A \text{ or } w \models B \\ w \models A \rightarrow B & \text{iff } w \not\models A \text{ or } w \models B \\ w \models \square A & \text{iff for all } v \in W, (w, v) \notin R \text{ or } v \models A \\ w \models \diamond A & \text{iff there exists some } v \in W, \text{ with } (w, v) \in R \text{ and } v \models A \end{array}$$

We say that w satisfies A iff $w \models A$ (without mentioning the valuation V). A formula A is called satisfiable in a model $\langle W, R, V \rangle$, iff there exists some $w \in W$, such that $w \models A$. A formula A is called satisfiable in a frame $\langle W, R \rangle$, iff there exists some valuation V and some world $w \in W$, such that $w \models A$.

A formula A is called valid in a model $\langle W, R, V \rangle$, written as $\langle W, R, V \rangle \models A$ iff it is true at every world in W . A formula A is valid in a frame $\langle W, R \rangle$, written as $\langle W, R \rangle \models A$ iff it is valid in all models $\langle W, R, V \rangle$.

Lemma 9 The operators \diamond and \square are dual, i.e. for all formulae A and all frames $\langle W, R \rangle$, the equivalence $\diamond A \leftrightarrow \neg \square \neg A$ holds.

Axiomatics

The simplest modal logic is called K and is given by the following axioms:

- All classical tautologies (and substitutions thereof)
- Modal Axioms: All formulae of the form $\Box(X \rightarrow Y) \rightarrow (\Box X \rightarrow \Box Y)$

and the inference rules

- Modus Ponens Rule: Conclude Y from X and $X \rightarrow Y$
- Necessitation Rule: Conclude $\Box X$ from X

A K derivation of X from a set S of formulae is a sequence of formulae, ending with X , each of it is an axiom of K , a member of S or follows from earlier terms by application of an inference rule. A K proof of X is a K derivation of X from \emptyset .

As an example take the K proof of $(\Box P \wedge \Box Q) \rightarrow \Box(P \wedge Q)$:

Tautology:	$P \rightarrow (Q \rightarrow (P \wedge Q))$
Necessitation:	$\Box(P \rightarrow (Q \rightarrow (P \wedge Q)))$
Modal axiom:	$\Box(P \rightarrow (Q \rightarrow (P \wedge Q))) \rightarrow (\Box P \rightarrow \Box(Q \rightarrow (P \wedge Q)))$
Modus ponens:	$\Box P \rightarrow \Box(Q \rightarrow (P \wedge Q))$
Modal axiom:	$\Box(Q \rightarrow (P \wedge Q)) \rightarrow (\Box Q \rightarrow \Box(P \wedge Q))$
Classical arg:	$\Box P \rightarrow (\Box Q \rightarrow \Box(P \wedge Q))$
Classical arg:	$(\Box P \wedge \Box Q) \rightarrow \Box(P \wedge Q)$

There is a similar proof of the converse of this implication; hence it follows that in K we have

$$\Box(P \wedge Q) \leftrightarrow (\Box P \wedge \Box Q)$$

Note that distributivity over disjunction does not hold! (Why?)

Extensions of K

Starting from the modal logic K one can add additional axioms, yielding different logics. We list the following basic axioms:

$$K : \Box(X \rightarrow Y) \rightarrow (\Box X \rightarrow \Box Y)$$

$$T : \Box A \rightarrow A$$

$$D : \Box A \rightarrow \Diamond A$$

$$4 : \Box A \rightarrow \Box \Box A$$

$$5 : \Diamond A \rightarrow \Box \Diamond A$$

$$B : A \rightarrow \Box \Diamond A$$

Traditionally, if one adds axioms A_1, \dots, A_n to the logic K one calls the resulting logic $KA_1 \dots A_n$. Sometimes, however the logic is so well known, that it is referred to under another name; e.g. $KT4$, is called $S4$.

These logics can as well be characterised by certain classes of frames, because it is known that particular axioms correspond to particular restrictions on the reachability relation R of the frame. If $\langle W, R \rangle$ is a frame, then a certain axiom will be valid on $\langle W, R \rangle$, if and only if R meets a certain restriction. Some restrictions are expressible by first-order logical formulae where the binary predicate $R(x, y)$ represents the reachability relation:

- T : Reflexive $\quad \forall w : R(w, w)$
- D : Serial $\quad \forall w \exists w' : R(w, w')$
- 4 : Transitive $\quad \forall s, t, u : (R(s, t) \wedge R(t, u)) \rightarrow R(s, u)$
- 5 : Euclidean $\quad \forall s, t, u : (R(s, t) \wedge R(s, u)) \rightarrow R(t, u)$
- B : Symmetric $\quad \forall w, w' : R(w, w') \rightarrow R(w', w)$

Multimodal Logics – An example

If modal logics is to be used for expressing the knowledge of various agents, one needs operators which are parametrised: we introduce box operators $[a]$, which are parametrised by arbitrary terms a ; the same holds for $\langle a \rangle$, the parametrised diamond operator.

A king, wishing to know which of his three advisers is the wisest, paints a white spot on each of their foreheads, tells them the spots are black or white and that there is at least one white spot, and asks them to tell him the color of their own spots. After a time the first wise man says, “I do not know whether I have a white spot.” The second, hearing this, also says he does not know. The third (truly!) wise man then responds, “My spot must be white.”

The following is a formalisation using the the abbreviation $\Box F$, which stands for arbitrary nested parametrised \Box -operators. If we had only 2 wise man A and B , $\Box F$ would stand for $[A]F \wedge [B]F \wedge [A][B]F \wedge [B][A]F \wedge [A][A]F \dots$. Hence in general $\Box F$ stands for “ F is generally known”.

- $B \neq A \wedge C \neq A \wedge C \neq B$ the three wise are different
- $\Box(w(A) \vee w(B) \vee w(C))$ it is generally known, that one of them has a white spot
- $\Box(\forall x, y : x \neq y \rightarrow (\neg w(x) \rightarrow [y]\neg w(x)))$ it is generally known, that if someone has no white spot, the others know it
- $[C][B]\neg[A]w(A)$ C knows, that B knows, that A does not know the colour of his spot
- $[C]\neg[B]w(B)$ C knows, that B does not know the colour of his spot

The theorem to be proven is “ C knows that he has a white spot”:

$$[C]w(c)$$

If we had a theorem prover for modal logics we could apply it to the above specified problem in order to obtain a proof and hence an explanation for the theorem.

In the rest of this section we will introduce two methods for the definition of such a theorem prover: A direct tableau calculus and a translation method in to first order classical predicate logic.

Modal Logic Tableaux

In classical propositional logics we introduced a tableau calculus (cf Definition) for a logic L as a finitely branching tree whose nodes are formulas from L .

Given a set Φ of formulae from L , a tableau for Φ was constructed by a (possibly infinite) sequence of applications of a tableau rule schema:

$$\rho \frac{\psi}{D_1 \mid D_2 \mid \cdots \mid D_n}$$

where the premise ψ as well as the denominators D_1, \dots, D_n are sets of formulae; ρ is the name of the rule.

We introduce K -tableau with the help of the following rules:

$$(\wedge) \frac{X; P \wedge Q}{X; P; Q} \quad (\vee) \frac{X; \neg(P \wedge Q)}{X; \neg P; \mid X; \neg Q}$$

$$(\perp) \frac{X; P; \neg P}{\perp} \quad (\neg) \frac{X; \neg\neg P}{X; P}$$

$$(\theta) \frac{X; Y}{X} \quad (K) \frac{\Box X; \neg\Box P}{X; \neg P}$$

A tableau for a set X of formulae is a finite tree with root X whose nodes carry finite formulae sets. The rules for extending a tableau are given by:

- choose a leaf node n with label Y , where n is not an end node, and choose a rule ρ , which is applicable to n ;
- if ρ has k denominators then create k successor nodes for n , with successor i carrying an appropriate instance of denominator D_i ;
- if a successor s carries a set Z and Z has already appeared on the branch from the root to s then s is an end node.

A branch is called closed if its end node is carrying \perp , otherwise it is open.

As in the classical case, a formula A is a theorem in modal logic K , iff there is a closed K -tableau for the set $\neg A$.

As an example take the formula $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$: its negation is certainly unsatisfiable, because the formula is an instance of our previously given K -axiom.

Some remarks are in order:

- The θ -rule, called the thinning rule, is necessary in order to construct the premisses for the application of the K -rule.
- Both can be combined by a new rule

$$(K\theta) \frac{Y; \Box X; \neg \Box P}{X; \neg P}$$

- In order to get tableau calculi for the other mentioned calculi, like T and $K4$ one has to introduce additional rules:

$$(T) \frac{X; \neg \Box P}{X; \Box P; P}$$

which obviously reflects reflexivity of the reachability relation, or

$$(K4) \frac{\Box X; \neg \Box P}{X; \Box X; \neg P}$$

reflecting transitivity.

Translation Method

There are several methods aiming at a translation of propositional modal logics into first order predicate logics.

The idea is, to transform the semantic conditions for the reachability into logical formulae: One rule for the definition of the semantic was:

$$w \models \Box A \quad \text{iff for all } v \in W, (w, v) \in R \text{ or } v \models A$$

This can be compiled into a formula by substituting the modal formula $\Box P$ by the first order formula $\forall y R(x, y) \rightarrow P'(y)$. Hence we can eliminate all modal operators by introducing the first order translations. The result of such a translation is a classical first order formula, which can be processed by the methods we have seen before.

For a modal formula F we define its translation F^* :

- $P^* = P(x)$, if P is a propositional constant
- $(\neg F)^* = \neg(F^*)$
- $(F \wedge G)^* = (F^* \wedge G^*)$

- $(\Box F)^* = (\forall y(R(x, y) \rightarrow F^*(x/y)))$, where y is a new variable not occurring in F^* and $F^*(x/y)$ is the result of replacing all free occurrences of x in F^* by y .

As a result we have

Theorem 25 *F is a valid modal formal in K iff F^* is a valid first order formula.*

Together with the observation that validity in modal logic K (like in many others) is decidable, we hence have a sublogic of first order classical predicate logic which is decidable! Modal logic can be seen as a fragment of 2-variable first-order logic FO^2 .

Temporal Logics

The two modalities \Box and \diamond cannot be used to distinguish between past and future. For this we need a multi-modal logic with the following \Box -operators

- $[F]A$: A holds always in the future
- $[P]A$: A holds always in the past
- $[A]A$: A holds always

and the corresponding \diamond -operators:

- $\langle F \rangle A$: A holds somewhere in the future
- $\langle P \rangle A$: A holds somewhere in the past
- $\langle A \rangle A$: A holds somewhere

The semantics is then given as before, by giving constraints for the three reachability relations or by giving appropriate axioms, e.g.

- $[F]A \rightarrow [F][F]A$: Transitivity; an analog axiom holds for the two other \Box -operators.
- $A \rightarrow [F]\langle P \rangle A$: if we go from a time point t in the future t' , we can go back in the past to the time point where A was true.
- $[A]A \leftrightarrow [F]A \wedge [P]A$: connection of past with future.

In addition there are many other aspects of temporal logics. E.g. one can distinguish between left- and rightlinear structures or between dense and discrete time structures.

Current Propositional Formulas

2

This is your set of current propositional formulas:

Currently empty.

²Interactions developed by: P. Baumgartner, A. Deutsch, M. Maron, C. Obermaier, A. Sinner

Truth Tables

Here are the current truth tables (possibly from a previous run):
Currently empty.

Tableaux

Here is the current tableau (possibly from a previous run):
Currently none.

Current Propositional Clauses

This is your set of current propositional clauses:

Currently empty.

Exercises

Current exercise:

Transform the following formula into CNF:

Currently empty. How to do the exercise:

Current transformations: Currently empty.

Instantiated completeness proof

This is your set S of current clauses : $\{\}$

Unfortunately S is not unsatisfiable, so it does not make sense to explain the proof with it. Please try again with an unsatisfiable set of clauses.

Resolution Closure

This is your set Res^* (current clauses), (possibly from a previous run):

Currently empty.



Labeling algorithm

This is the result, (possibly from a previous run):

Currently none.

CNF Tableaux

Here is the current tableau (possibly from a previous run):
Currently none.

Otter

This is the result, (possibly from a previous run):
Currently none.

Protein

This is the result, (possibly from a previous run):
Currently none.

Current Predicate Logic Formulas

This is your set of current predicate logic formulas:

Currently empty.

Current Predicate Logic Clauses

This is your set of current predicate logic clauses:

Currently empty.

Otter

[Apply Otter to the current predicate logic clauses](#) [Delete current Otter Result](#)

This is the result, (possibly from a previous run):
Currently none.

Protein

Apply PROTEIN to the current clauses Delete current Protein result This is the result, (possibly from a previous run):
Currently none.

Current Terms

This is your set of current terms:

Currently empty.

Current terms manipulation:

Enter some terms:

Help:

[Add to](#) / [Overwrite](#) current terms

Save/Reload current terms:

[Save](#) to database (enter name):

[Delete select](#)

[Reload](#) from

[Reload](#) from

Unification

[Unify selected](#) / [Unify all](#) terms.



This is the result, (possibly from a previous run): Currently none.

Layout

Format document for screen