# CSC375/CSCM75
# Logic for Computer Science

Ulrich Berger

Department of Computer Science
Swansea University

Fall 2017

u.berger@swan.ac.uk
http://www.cs.swan.ac.uk/∼csulrich/

tel 513380, fax 295708, room 306, Faraday Tower

## 1 Introduction

The aim of this course is to give the student a working knowledge in logic and its applications in Computer Science. Rather than trying to cover logic in its full breadth, which, given the diversity and complexity of the subject, would be an impossible task for a one-semester course, we will focus on the most fundamental concepts which every computer scientists should be familiar with, be it as a practitioner in industry or as a researcher at university. Although the selection of topics reflects to some extent my personal view and taste of logic, I will aim at a balanced presentation that also takes into account aspects of logic are not so close to my own research interests.

Throughout the course I will emphasize that logic is something one *does*: one *models* a computing system, *specifies* a data type or a program, *checks* or *proves* that a system satisfies a certain property. We will do a lot of exercises where these logical activities will be trained, mostly on the blackboard or with pencil and paper, and occasionally using the computer. The aim is to provide the student with active logic skills to solve computing problems.

Besides its practical applications we will also look into the historical development of logic. What were the main philosophical and mathematical questions that led to modern logic as we know it today? We will see that these questions and their answers did shape not only logic, but also large parts of Computer Science.

The main *prerequisites* for this course are *curiosity* and a *good appetite for solving problems*. No prior knowledge of logic is required. However, I take it for granted that the student is able to digest formal definitions of syntactic entities such as terms or formulas. For a Computer Science student, who is familiar with formal syntactic objects such as computer programs, this will not be a problem at all.

"Logic for Computer Science" is a stand-alone course, but it is also intended to support other Computer Science modules offered, in particular the modules on Embedded Systems, High Integrity Systems, Software Testing, Big Data and Machine Learning, and Modeling and Verification Techniques. Our course does not follow a particular textbook, but it will use material from several books and lectures on logic. Some of the main sources are:

[1] D van Dalen, Logic and Structure, 3rd edition, Springer, 1994.

[2] J H Gallier, Logic for Computer Science, John Wiley & Sons, 1987.

[3] Handbook of Logic in Computer Science, Vol. 1-6, S Abramsky, D M Gabbay, T S E Maibaum, eds, OUP, 1994.

[4] J R Shoenfield, Mathematical Logic, Addison-Wesley, 1967.

[5] D B Plummer, J Barwise, J Etchemendy, Tarski's World, CSLI Lecture Notes, 2008.

[6] U Schoening, Logic for Computer Science, Birkhäuser, 1989.

[7] A S Troelstra, D van Dalen, Constructivism in Mathematics, Vol. I, North-Holland, 1988.

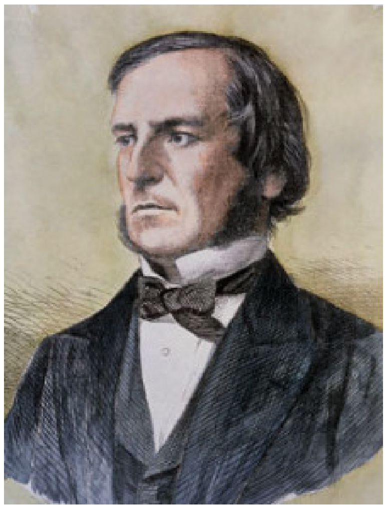
[8] D Velleman, How to Prove It, 2nd edition, CUP, 1994.

[9] B, Programming with Abstract Data Types, Lecture Notes, Swansea University, 2009.

[10] Michael Huth, Mark Ryan, Logic in Computer Science, Cambridge University Press, 2004.

These are preliminary course notes which are being developed as the course progresses. I will be grateful for reporting to me any errors or suggestions for improvements. For students seriously interested in logic I recommend to study in addition the sources listed below as well as further literature on logic which will be recommended during the course.

# 2 Propositional Logic


George Boole (1815 - 1864)
English Mathematician, Philosopher and Logician

Propositional Logic is the logic of *atomic propositions* that are combined by *logical connectives* such as “and”, “or”, “not”, “implies”.

Here is an example taken from [2]. Consider the following propositions

$T$ “John is a teacher”,

$R$ “John is rich”,

$S$ “John is a rock singer”.

We call $T$, $R$, $S$, *atomic propositions.* Now consider the following statements:

- John is a teacher.
- It is not the case that John is a teacher and John is rich.
- If John is a rock singer then John is rich.

We wish to show that the above assumptions imply

It is not the case that John is a rock singer.

Using the abbreviations and parentheses to disambiguate the statements (in particular the second assumption!) this amounts to showing that the following statement holds:

(∗) ($T$ and not($T$ and $R$) and ($S$ implies $R$)) implies not($S$)

holds. To keep the notation of statements short we first introduce some common symbols for the logical connectives:

| | | |
|---|---|---|
| $\wedge$ | for | "and" |
| $\to$ | for | "implies" |
| $\neg$ | for | "not" |

Now the statement we wish to prove can be written as follows:

(∗) $(T \wedge \neg(T \wedge R) \wedge (S \to R)) \to \neg S$

Without worrying at the moment too much about what it precisely means for a statement to "hold" we can give a proof of our statement using common sense reasoning:

To prove (∗) we assume that the premises, $T$, $\neg(T \wedge R)$ and $S \to R$ are all true.

We must show that the conclusion, $\neg S$, is true, that is, $S$ is false. In other words, we must show that assuming $S$ leads to a contradiction.

Hence, we assume $S$, aiming for a contradiction.

Since we assumed $S$ and we assumed $(S \to R)$ we know that $R$ holds. Hence in fact $(T \wedge R)$ holds since we assumed $T$. But this contradicts the assumption $\neg(T \wedge R)$.

It is possible to make this kind of reasoning completely precise and formal. Moreover, there are computer systems carrying out the reasoning automatically. We will explore this possibility later in the course.

Yet, there is another, completely different method of finding out that the statement (∗) holds: Since we know nothing about the atomic propositions $T$, $R$, $S$, other than that they are either true or false, we can simply try all possible ways of assigning "true" or "false" to $T$, $R$, $S$ and check that statement (∗) is true in all cases. It is rather obvious that this method can be automatised as well.

Note that the method of checking all assignments relies on the assumption that the atomic propositions involved are completely independent of each other and that it only matters whether they are true or false, but not what they actually "mean". In other words, in propositional logic atomic propositions are viewed as "black boxes" whose internal structure is invisible.

To highlight this point consider the propositions

$$\begin{array}{rcl} A & := & \text{it is raining} \\ B & := & \text{the street is wet} \end{array}$$

Common sense tells us that the statement $A \to B$ holds, but there is no way finding this out using a logical proof or the assignment method. One can say that the statement $A \to B$ holds, however not for reasons of propositional logic, but for reasons having to do with the *meaning* of $A$ and $B$.

On the other hand our proof above was *purely logical*. It did not depend on the meanings of the atomic statements $T$, $R$ and $S$.

## 2.1 The three elements of logic

Although our example of teacher John is very simple, it suffices to discern *three fundamental elements of logic*:

**Syntax** The expression $(*)$ $(T \wedge \neg(T \wedge R) \wedge (S \to R)) \to \neg S$ is an example of a *formula*, that is, a formal syntactic representation of a statement. In a given logic it will always be precisely defined which expressions are wellformed, that is, syntactically correct, formulas. Different logics may have different formulas. In Computer Science, formulas are used to *specify* computer programs, processes, etc., that is, to precisely describe their properties.

**Semantics** Given a formula, the question arises what is its *semantics*, that is, *meaning*, or, more specifically, what does it mean for a formula to be *true*? For example, looking at the formula $\neg(T \wedge R)$ we might say that we cannot determine its truth unless we know whether $T$ and $R$ are true. This means that we can determine the truth of this formula only with respect to a certain *assignment of truth values*. Such an assignment is a special case of a *structure*, also called *model*, or *world*. In semantics one often studies structures and the question whether a formula is true in a structure. However, looking at formula $(*)$ we might say that it is true irrespectively of the model. One also says $(*)$ is *valid*, or *logically true*.

**Proof** How can we find out whether a formula is true? This question can be asked with respect to a given model, if one wants to know whether the formula holds in the given model. But one can also ask whether the formula is true in *all* models, that is, whether it is logically true? Are there mechanisable, or even automatisable methods to *prove* that a formula is true? In Computer Science, a lot of research effort is spent on developing systems that can automatically or semi-automatically prove formulas or construct models where a formula is false. Many of these systems are already in use to improve the quality and dependability of safety critical computer applications.

All forms of logic are essentially concerned with the study of these three elements: syntax, semantics, proof. They will also be the guiding principles of this logic course.

## 2.2 Syntax of propositional logic

We assume we are given an infinite sequence $P_1, P_2, \ldots$ of *atomic propositions.*

### 2.2.1 Propositional formula

The set of *propositional formulas* is defined inductively as follows.

(i) Every atomic proposition is a propositional formula.

(ii) $\top$ and $\bot$ are propositional formulas.

(iii) If $A$ is a propositional formula, then $\neg A$ is a propositional formula .

(iv) If $A$ and $B$ are propositional formulas, then $(A \wedge B)$, $(A \vee B)$, and $(A \to B)$ are propositional formulas.

Some comments regarding:

- The phrase "defined inductively" used above means that propositional formulas are exactly those expressions that can be generated by the rules (i) - (iv).
- Atomic propositions are also called *propositional variables* (because, as we will see, their truth value may vary).
- The formulas $\top$ and $\bot$ represent the truth values "true" and "false". These are also often denoted by the numbers 1 and 0.
- Atomic propositions as well as $\top$ and $\bot$ are called *basic formulas.* All other formulas are called *composite formulas.*
- The formula $\neg A$ is called the *negation* of $A$, pronounced "not $A$". Some textbooks use the notation $\sim A$ instead.
- The formula $(A \wedge B)$ is called *conjunction* of $A$ and $G$, pronounced "$A$ and $B$".
- The formula $(A \vee B)$ is called *disjunction* of $A$ and $B$, pronounced "$A$ or $B$".
- A formula of the form $(A \to B)$ is called *implication*, pronounced "$A$ implies $B$". In some textbooks implication is written $A \supset B$.
- It is often convenient to introduce the abbreviation

  $$(A \leftrightarrow B) := (A \to B) \wedge (A \to B)$$

  A formula of the form $(A \leftrightarrow B)$ is called *equivalence*, pronounced "$A$ is equivalent to $B$". In many textbooks this is written $A \equiv B$.

- The symbols $\neg, \wedge, \vee, \rightarrow$ are called "propositional connectives" (or "logical connectives").

Since in this and the following chapters we will only consider propositional formulas and no other kind of formulas we will allow ourselves to say "formula" instead of "propositional formula".

Definition 2.2.1 maybe equivalently formulated as a context free grammar in Backus-Naur form:

$$\begin{array}{rcl} <formula> & ::= & <atomic\ proposition> \\ & \mid & \top \\ & \mid & \bot \\ & \mid & \neg <formula> \\ & \mid & (<formula> \wedge <formula>) \\ & \mid & (<formula> \vee <formula>) \\ & \mid & (<formula> \rightarrow <formula>) \end{array}$$

### 2.2.2 Examples of formulas

Let $A, B, C$ be atomic propositions.

(a) $(A \vee \neg A)$

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

(c) $((A \vee B) \rightarrow (A \wedge B))$

(d) $(((A \vee B) \rightarrow (A \wedge B)) \rightarrow \bot)$

(e) $((A \vee B) \rightarrow ((A \wedge B) \rightarrow \bot))$

These examples, in particular (d) and (e), demonstrate that a proliferation of parentheses can make formulas hard to read. In order to improve readability we will make use of the following conventions:

- Outermost parentheses are omitted. For example, we write $\neg A \wedge B$ instead of $(\neg A \wedge B)$.
- Implication is the least binding operator. For example, we write $A \wedge B \rightarrow B \vee A$ for $(A \wedge B) \rightarrow (B \vee A)$.

- Implication is assumed to associate to the right. This means, for example, that we write $A \to B \to C \to D$ for $A \to (B \to (C \to D))$.
- Conjunction and disjunction are assumed to associate to the left. For example, we write $A \land B \land C$ instead of $(A \land B) \land C$.

Warnings: We do *not* assume that $\land$ binds stronger than $\lor$, and, for example, in the formula $(A \to B) \to C$ the parentheses may *not* be omitted. Furthermore, the formulas $\neg A \land B$ and $\neg(A \land B)$ are *not* the same. In order to distinguish the latter two formulas more clearly we may occasionally enclose a negation in (redundant) parentheses, as, for example, in $(\neg A) \land B$.

### 2.2.3 Exercise

(a) Write down the formulas of Example 2.2.2 omitting as many parentheses as possible according to the conventions above.

(b) Add to the formula $A \land B \to A \lor C \to D$ all omitted parentheses.

(c) Is the expression $A \land B \lor C$ a correct denotation of a formula?

### 2.2.4 Exercise

Write down the formula of the initial exercise about John the teacher, with maximal and with minimal parentheses.

## 2.3 Induction on formulas

The way formulas are defined in 2.2.1 is an example of an *inductive definition*. Instead of going into the theory of this kind of definition I try to give the intuition. The clauses (i) - (iv) are to be viewed as a *generation process*: All expression, and only those, which can be generated by a finite number of applications of the clauses are formulas.

Associated with this inductive definition is a principle of *proof by induction on formulas*. It says that in order to prove that a certain property $P(A)$ holds for all formulas $A$ it is enough to prove

- **Induction base:** $P(A)$ holds if $A$ is a basic formula.
- **Induction step:** $P$ holds for a composite formula provided $P$ holds for its parts. For example when proving $P(A \to B)$ one may assume (as induction hypothesis) that $P(A)$ and $P(B)$ hold (similarly for the other logical connectives).

### 2.3.1 Example of a proof by induction on formulas

**Lemma 1.** Every formula has as many opening parentheses as closing parentheses (assuming all parentheses are written).

**Proof.** By induction on formulas. We set

$$P(A) := A \text{ has as many opening parentheses as closing parentheses}$$

*Induction base*: Clearly $P$ holds for basic formulas, since these have no parentheses at all.

*Induction step*: Consider a composite formula, for example, $(A \wedge B)$. By induction hypothesis $A$ and $B$ both satisfy $P$. Hence clearly $(A \wedge B)$ satisfies $P$ as well since exactly on opening and one closing parentheses were added. For the other logical connectives the argument is similar.

This completes the proof of the lemma.

Induction of formulas is very similar to the well-known *induction on natural numbers* which is a proof principle saying that in order to prove a property $P(n)$ for all natural number $n$ it suffices to prove

- the base case, $P(0)$,
- and the step, $P(n+1)$, under the assumption that $P(n)$ (induction hypothesis) holds.

The reason why induction on natural numbers is correct is the same as for formulas: natural numbers are inductively defined by the clauses

(i) 0 is a natural number.

(ii) If $n$ is a natural number, then $n + 1$ is a natural number.

## 2.4 Recursion on formulas

In addition to the fact that all formulas are generated by the clauses (i)-(iv) of Definition 2.2.1 it is true that this generation is non-ambiguous. That is, for each formula there is only one way to generate it. In other words each string representing a formula can be parsed in exactly one way. One says that formulas are *freely generated* by the clauses (i)-(iv).

Associated with the free generation of formulas is a principle of *definition by recursion on formulas*. This principle says that in order to define a function $f$ on all formulas it is enough to

- **Base case:** define $f(A)$ for all basic formulas $A$,
- **Recursive case:** define $f$ for a composite formula using the definition of for its parts. For example when defining $f(A \to B)$ one may recursively call $f(A)$ and $f(B)$.

### 2.4.1 Example of a function defined by recursion on formulas

We define the *depth* of a formula as follows

- $\mathrm{depth}(A) = 0$ for every basic formula.
- $\mathrm{depth}(\neg A) = \mathrm{depth}(A) + 1$.
- $\mathrm{depth}(A \diamond B) = \max(\mathrm{depth}(A), \mathrm{depth}(B)) + 1$ for $\diamond \in \{\wedge, \vee, \to\}$.

We also let $\mathrm{con}(A)$ be the number of occurrences of logical connectives $(\wedge, \vee, \to, \neg)$ in the formula $A$.

**Lemma 2.** $\mathrm{con}(A) < 2^{\mathrm{depth}(A)}$.

**Proof.** By induction on formulas.

Induction base: If $A$ is a basic formula, then $\mathrm{con}(A) = 0 < 1 = 2^0 = 2^{\mathrm{depth}(A)}$.

Induction step: We have to consider composite formulas.

$\neg A$: By induction hypothesis (i.h.) we have $\mathrm{con}(A) < 2^{\mathrm{depth}(A)}$. Therefore

$$\begin{array}{rcl} \mathrm{con}(\neg A) & = & \mathrm{con}(A) + 1 \\ & \overset{\text{i.h.}}{<} & 2^{\mathrm{depth}(A)} + 1 \\ & \leq & 2 * 2^{\mathrm{depth}(A)} \\ & = & 2^{\mathrm{depth}(A)+1} \\ & = & 2^{\mathrm{depth}(\neg A)} \end{array}$$

$A \diamond B$ where $\diamond \in \{\wedge, \vee, \to\}$: By induction hypothesis we have $\mathrm{con}(A) < 2^{\mathrm{depth}(A)}$ and $\mathrm{con}(B) < 2^{\mathrm{depth}(B)}$.

$$\begin{array}{rcl} \mathrm{con}(A \diamond B) & = & \mathrm{con}(A) + \mathrm{con}(B) + 1 \\ & < & \mathrm{con}(A) + 1 + \mathrm{con}(B) + 1 \\ & \overset{\text{i.h.}}{\leq} & 2^{\mathrm{depth}(A)} + 2^{\mathrm{depth}(B)} \\ & \leq & 2 * 2^{\max(\mathrm{depth}(A), \mathrm{depth}(B))} \\ & = & 2^{\mathrm{depth}(A \diamond B)} \end{array}$$

This completes the proof.

Natural numbers are also freely generated, namely from 0 and the successor operation. Consequently, there is a corresponding recursion principle which is called *primitive recursion*: in order to define a function $f$ on natural numbers it is enough to define

- $f(0)$ and
- $f(n+1)$ with a possible recursive call to $f(n)$.

Induction and recursion principles such as induction on formulas and natural numbers, recursion on formulas and primitive recursion occur very often in computer science and are of fundamental importance. It is therefore essential to be familiar with them and to be able to apply them. In fact, we will frequently use them in this course. These principles are instances of two general principles which are valid for all freely generated structures: *Structural Induction* and *Structural recursion* (see, for example the preliminary chapter of [2]).

**Recursion on formulas vs. general recursion** In most programming languages *general recursion* is allowed. This means that when defining a function $f$, a value $f(x)$ may be defined making recursive calls to $f(y)$ where $y$ is arbitrary.

Clearly, general recursion captures recursion on formulas, primitive recursion and, in general, structural recursion. So why bother with restricted forms of recursion?

The reason is that recursion on formulas (and more generally structural recursion) guarantees that the recursively defined function will *terminate* for all inputs, while with general recursion this is of course not the case. Moreover, it can be decided easily whether a given recursive definition is structural recursive or not, while the question whether or not a general recursive program terminates is undecidable, due to the *undecidability of the Halting Problem.*

### 2.4.2 Exercise

Prove $\mathrm{depth}(A) \leq \mathrm{con}(A)$.

### 2.4.3 Exercise

Give a definition of $\mathrm{con}(A)$ by recursion on formulas.

#### 2.4.4 Exercise

Compute depth$(A)$ and con$(A)$ for the following formulas $A$:

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

(b) $((A \to B) \to A) \to A$

#### 2.4.5 Exercise

Let atom$(A)$ be the number of occurrences of atomic formulas of $A$. Give a definition of atom$(A)$ by recursion on formulas.

#### 2.4.6 Exercise

Prove atom$(A) \leq 2 \cdot \mathrm{con}(A) + 1$.

#### 2.4.7 Exercise

Define a Haskell data type of propositional formulas.

#### 2.4.8 Exercise

Implement the functions depth$(A)$, con$(A)$ and atom$(A)$ in Haskell.

## 2.5 Semantics of propositional logic

The semantics (= meaning) of a formula $A$ is a *truth value*: either 0 (for "false") or 1 (for "true"). The truth values 0 and 1 are also called *boolean values.* Of course, the meaning of $A$ depends on the meaning of the atomic propositions occurring in $A$. Once the meaning of these atomic propositions is fixed, the meaning of $A$ can be computed according to the following truth tables of the propositional connectives.

| $\top$ |
|---|
| 1 |

| $\bot$ |
|---|
| 0 |

| $x$ | $\neg x$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $x$ | $y$ | $x \vee y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $x$ | $y$ | $x \to y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

These tables are to be understood as shorthand notations for the definitions of *Boolean functions* associated with the propositional connectives:

$$\neg : \{0,1\} \to \{0,1\}, \qquad \wedge, \vee, \to : \{0,1\}^2 \to \{0,1\}$$

For example,

$$\begin{aligned} 0 \to 0 &= 1 \\ 0 \to 1 &= 1 \\ 1 \to 0 &= 0 \\ 1 \to 1 &= 1 \end{aligned}$$

### 2.5.1 Exercise (from aptitude test for Foundation Degree)

There are four cards on the table:

- Card 1 is from the red deck
- Card 2 is an Ace of Spade
- Card 3 is from the blue deck
- Card 4 is a Three of Heart

Claim: Every Ace amongst these four cards comes from the blue deck.

Which cards need be turned over in order to be certain that the claim is true?

**Solution:** Let $A$ be the statement "the card is an Ace" and $B$ be the statement "the card is from the blue deck".

The Claim states that the formula $A \to B$ holds for all four cards. Therefore, we need to turn over those cards for which this formula might be false.

According to the truth table of implication, the formula $A \to B$ is false if and only if $A$ is true and $B$ is false, that is, if the card is an Ace but not from the blue deck.

Now check for which of the cards this might be the case.

In the following we fix a finite set of atomic propositions

$$\mathcal{A} := \{P_1, \ldots, P_n\}$$

and consider only formulas containing atomic propositions from $\mathcal{A}$.

### 2.5.2 Definition (Assignment)

An *assignment* is a mapping $\alpha : \mathcal{A} \to \{0,1\}$. Hence, $\alpha$ assigns to each atomic proposition $P_i \in \mathcal{A}$ a truth value $\alpha(P_i) \in \{0,1\}$. An assignment $\alpha$ can also be viewed as an array of boolean values $(b_1, \ldots, b_n)$ where $\alpha(P_i) = b_i$ for $i \in \{1, \ldots, n\}$.

For example, the array $(1,0,1)$ denotes the assignment $\alpha$ with $\alpha(P_1) = 1$, $\alpha(P_2) = 0$ and $\alpha(P_3) = 1$.

### 2.5.3 Definition (truth value of a formula)

We define the *truth value of $A$ under an assignment $\alpha$*

$$[\![A]\!]\alpha \in \{0,1\}$$

by recursion on formulas:

$$\begin{array}{rcll} [\![P_i]\!]\alpha & = & \alpha(P_i) & \\ [\![\top]\!]\alpha & = & 1 & \\ [\![\bot]\!]\alpha & = & 0 & \\ [\![\neg A]\!]\alpha & = & \neg([\![A]\!]\alpha) & \\ [\![A \diamond G]\!]\alpha & = & [\![A]\!]\alpha \diamond [\![G]\!]\alpha & \quad \text{for } \diamond \in \{\wedge, \vee, \to\} \end{array}$$

### 2.5.4 Definition (substitution)

Let $A, B$ be formulas and $P$ an atomic proposition. By $A[P := B]$ we denote the formula obtained by substituting (that is, replacing) in $A$ every occurrence of $P$ by $B$.

For example, if $A = P \to P$ and $B = C \wedge P$, then $A[P := B] = C \wedge P \to C \wedge P$.

### 2.5.5 Exercise

Define $A[P := B]$ by recursion on the formula $A$.

**Solution.**

$$\begin{array}{rcll}
P[P := B] & = & B & \\
Q[P := B] & = & Q & \text{if } Q \text{ is an atomic proposition different from } P \\
\top[P := B] & = & \top & \\
\bot[P := B] & = & \bot & \\
(\neg A)[P := B] & = & \neg(A[P := B]) & \\
(A \diamond A')[P := B] & = & A[P := B] \diamond A'[P := B] & \text{for } \diamond \in \{\wedge, \vee, \rightarrow\}
\end{array}$$

### 2.5.6 Substitution Theorem

Let $A, B$ be formulas and $P$ an atomic proposition. For every assignment $\alpha$ we have

$$[\![A[P := B]]\!]\alpha = [\![A]\!]\alpha'$$

where $\alpha'$ is the assignment that is identical to $\alpha$ except that $\alpha'(P) := [\![B]\!]\alpha$.

**Proof.** By induction on formulas. We refer to the recursive definition of substitution given in the solution to Exercise 2.5.5.

$[\![P[P := B]]\!]\alpha = [\![B]\!]\alpha = \alpha'(P) = [\![P]\!]\alpha'$

If $A$ is an atomic proposition different from $P$, then
$[\![A[P := B]]\!]\alpha = [\![A]\!]\alpha = \alpha(A) = \alpha'(A) = [\![A]\!]\alpha'$

$$\begin{array}{rcll}
[\![\neg A[P := B]]\!]\alpha & = & [\![\neg A[P := B]]\!]\alpha & \text{by definition of substitution} \\
& = & \neg([\![A[P := B]]\!]\alpha) & \text{by definition of the value of a formula} \\
& = & \neg([\![A]\!]\alpha') & \text{by induction hypothesis} \\
& = & [\![\neg A]\!]\alpha' & \text{by definition of the truth value of a formula}
\end{array}$$

The proof for the remaining cases (conjunction, disjunction, ...) is similar.

**Remark.** The assignment $\alpha'$ in the Substitution Theorem is often denoted by $\alpha[P := b]$ where $b := [\![B]\!]\alpha \in \{0,1\}$. In general the "updated" assignment $\alpha[P := b]$ is defined as

$$\begin{array}{lll} \alpha[P := b](P) & := & b \\ \alpha[P := b](Q) & := & \alpha(Q) \quad \text{if } Q \text{ is an atomic proposition different from } P \end{array}$$

With this notation the Substitution Theorem reads

$$[\![A[P := B]]\!]\alpha = [\![A]\!](\alpha[P := [\![B]\!]\alpha])$$

**Remark.** The Substitution Theorem says that the value of a formula which contains a formula $B$ as a subformula only depends on the *value* of $B$ but not on its syntactic form. The principle expressed by this theorem is called

*Referential Transparency*

Referential Transparency also holds for arithmetic expressions, derivations, types, and many other types of syntax with a well defined semantics.

Referential Transparency is essential for any kind of reasoning about syntax and semantics because it allows to reason in a modular way. For example, in a formula $A$ we may replace a subformula $B$ by an equivalent formula $B'$ without changing the meaning of $A$ (this is shown in the Equivalence Theorem 2.6.8 below; what it means for two formulas to be equivalent is defined in 2.6.4).

Referential Transparency does hold for functional programs, but generally not for programs written in an imperative language such as C or Java. The reason is that the Substitution Theorem may fail if the substituted program performs side effects. Therefore, reasoning about imperative programs is more difficult than reasoning about functional programs.

## 2.6 Logic gates and equivalence

In Sect. 2.5 we defined the meaning of the propositional connectives as certain boolean functions $\neg : \{0,1\} \to \{0,1\}$, $\wedge, \vee, \to : \{0,1\}^2 \to \{0,1\}$. Now we study boolean functions in general.

### 2.6.1 Definition (logic gate, boolean function)

A *logic gate* or called *boolean function* or *switch*) of $n$ arguments is a function

$$f : \{0,1\}^n \to \{0,1\}$$

### 2.6.2 Exercise

Obviously, there exist only finitely many logic gates of $n$ arguments. How many?

### 2.6.3 Definition (logic gate defined by a formula)

Since an assignment $\alpha$ can be viewed as an array of boolean values

$$\alpha = (\alpha(P_0), \ldots, \alpha(P_{n-1})) \in \{0,1\}^n$$

the value of a formula $A$ can also be viewed as a boolean function or logic gate

$$[\![A]\!] : \{0,1\}^n \to \{0,1\}, \quad \alpha \mapsto [\![A]\!]\alpha$$

### 2.6.4 Definition (equivalence)

Usually, one is only interested in the logic gate a formula defines. Therefore, two formulas $A$, $B$ are called *equivalent*, in symbols, $A \equiv B$ if they define the same logic gate, that is, $[\![A]\!]\alpha = [\![B]\!]\alpha$ for all assignments $\alpha$.

### 2.6.5 Exercise

Let $A$, $B$ be atomic formulas. Which of the following formulas are equivalent?

(a) $A \to \neg B$

(b) $\neg(B \wedge A)$

(c) $\neg B \to A$

Here is a list of important equivalences:

$$
\begin{array}{rcl}
A \wedge B & \equiv & B \wedge A \\
(A \wedge B) \wedge C & \equiv & A \wedge (B \wedge C) \\
A \vee B & \equiv & B \vee A \\
(A \vee B) \vee C & \equiv & A \vee (B \vee C) \\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C) \\
A \vee (B \wedge C) & \equiv & (A \vee B) \wedge (A \vee C) \\
\neg(A \wedge B) & \equiv & \neg A \vee \neg B \\
\neg(A \vee B) & \equiv & \neg A \wedge \neg B \\
\neg(A \to B) & \equiv & A \wedge \neg B \\
\neg\neg A & \equiv & A \\
\neg A & \equiv & A \to \bot \\
A \to B & \equiv & \neg A \vee B \\
A \wedge \neg A & \equiv & \bot \\
A \vee \neg A & \equiv & \top \\
(A \wedge B) \to C & \equiv & A \to (B \to C) \\
(A \vee B) \to C & \equiv & (A \to C) \wedge (B \to C) \\
(\neg A) \to B & \equiv & (\neg B) \to A
\end{array}
$$

### 2.6.6 Exercise

Verify that the equivalences above do indeed hold by checking in each case that the formula on left hand side and the formula on the right hand side have the same truth value for all assignments.

### 2.6.7 Exercise

For each of the following formulas find an equivalent one which is shorter:

(a) $A \wedge \top$

(b) $A \wedge \bot$

(c) $A \vee \top$

(d) $A \vee \bot$

(e) $A \to A$

(f) $A \to \top$

(g) $\top \to A$

(h) $\bot \to A$

(i) $\neg A \to B$

(j) $A \vee \neg A$

(k) $A \wedge \neg A$

(l) $((A \to B) \to A) \to A$

### 2.6.8 Equivalence Theorem

Let $A, A', B, B'$ be formulas and $P$ an atomic proposition.

If $A \equiv A'$ and $B \equiv B'$, then $A[P := B] \equiv A'[P := B']$.

**Proof.** This is an easy consequence of the Substitution Theorem (detailed proof left as an exercise for the lecture).

**Remark.** Since there are only finitely many logic gates of $n$ arguments, but infinitely many formulas with $n$ atomic propositions, each logic gate can be defined by many (in fact infinitely many) formulas. Finding for a given logic gate a shortest formula defining it is an important and computationally difficult problem. We will later look at some of the most important approaches to this problem.

## 2.7 Satisfiability and logical validity

### 2.7.1 Definition (satisfaction, model, tautology)

If $[\![A]\!]\alpha = 1$ we say $\alpha$ *satisfies* $A$, or $\alpha$ *is a model of* $A$, in symbols

$$\alpha \models A$$

A formula $A$ is called *satisfiable* if it has a model, otherwise it is *unsatisfiable*.

The *satisfiability problem* is the problem of deciding whether a formula is satisfiable.

A formula $A$ is called a *tautology* or *logically valid* if it is satisfied by all assignments, that is, $[\![A]\!]\alpha = 1$ for all assignments $\alpha$.

**Remarks.**

- Speaking in terms of logic gates, for a formula $A$ to be logically valid means that the logic gate $A$ it defines is constant 1. This is also the same as saying that $A \equiv \top$.
- Similarly, for $A$ to be a unsatisfiable means that the logic gate $A$ is constant 0 or that $A \equiv \bot$.
- In particular, all tautologies are equivalent and all unsatisfiable formulas are equivalent.
- Formulas $A$ that are neither tautologies nor unsatisfiable (which is the case for most formulas) define a logic gate $A$ which is not constant, that is, there are assignments $\alpha, \alpha'$ such that $[\![A]\!]\alpha = 1$ and $[\![A]\!]\alpha' = 0$.

### 2.7.2 Exercise

Let $A$, $B$ be atomic formulas. Which of the following formulas are satisfiable, unsatisfiable, tautologies?

(a) $A \vee \neg A$

(b) $A \wedge \neg A$

(c) $B \to \neg A$

### 2.7.3 Exercise

Is the formula of the "John the teacher" example a tautology?

### 2.7.4 Theorem

A formula $A$ is a tautology if and only if $\neg A$ is unsatisfiable.

**Proof.** The proof has two parts.

1. Assume $A$ is a tautology. We show that $\neg A$ is unsatisfiable. Let $\alpha$ be an assignment. We must show that $\alpha$ is not a model of $\neg A$, that is, $[\![\neg A]\!]\alpha = 0$:

$$\begin{array}{rcll} [\![\neg A]\!]\alpha & = & \neg([\![A]\!]\alpha) & \text{by definition of the value of a negated formula} \\ & = & \neg(1) & \text{because } A \text{ is a tautology} \\ & = & 0 & \end{array}$$

2. Assume $\neg A$ is unsatisfiable. We show that $A$ is a tautology. Let $\alpha$ be an assignment. We must show that $\alpha$ is a model of $A$, that is, $[\![A]\!]\alpha = 1$.

Since $\neg(\neg(b)) = b$ for all $b \in \{0,1\}$ we have

$$\begin{array}{rcll} [\![A]\!]\alpha & = & \neg(\neg([\![A]\!]\alpha)) & \\ & = & \neg([\![\neg A]\!]\alpha) & \text{by definition of the value of a negated formula} \\ & = & \neg(0) & \text{because } \neg A \text{ is unsatisfiable} \\ & = & 1 & \end{array}$$

### 2.7.5 Theorem (closure of tautologies under arbitrary substitution)

Let $A$ and $B$ be formulas and $P$ an atomic proposition.

If $A$ is a tautology, then $A[P := B]$ is a tautology.

**Proof.** This is another easy consequence of the Substitution Theorem. The proof is left as an exercise for the student.

### 2.7.6 Exercise

(a) Does the converse of Theorem 2.7.5 hold? That is, is the following true?

If $A[P := B]$ is a tautology, then $A$ is a tautology.

(b) Does Theorem 2.7.5 still hold if "is a tautology" is replaced by "is satisfiable"?

(c) Does Theorem 2.7.5 still hold if "is a tautology" is replaced by "is unsatisfiable"?

Justify your answers.

### 2.7.7 Exercise

Brown, Jones and Smith are suspected of Tax evasion. They testify under oath as follows:

> Brown: Jones is guilty and Smith is innocent.
>
> Jones: If Brown is guilty, then so is Smith.
>
> Smith: I am innocent, but at least one of the others is guilty.

(a) Assuming everybody told the truth, who is innocent/guilty?

(b) Assuming the innocent told the truth and the guilty lied, who is innocent/guilty?

First, try to find the answers by informal reasoning.

Then, use the atomic propositions $B$ (Brown is guilty), $J$ (Jones is guilty), $S$ (Smith is guilty), to write the statements by Brown, Jones and Smith as propositional formulas and confirm your informal answers by checking the logic gates defined by these statements.

Additional (easy) questions:

(c) Is it possible that everybody told the truth?

(d) Is it possible that everybody lied?

# 3 Normal forms

In this chapter we introduce special classes of "normal forms" of formulas which are important in many applications of logic in Computer Science. Some of these classes are still rich enough to define all logic gates, some are not. We will also look at compact representations of logic gates that are frequently used in practical applications.

## 3.1 Conjunctive Normal Form (CNF)

### 3.1.1 Definition (Literal)

A *literal* is an atomic proposition or the negation of an atomic proposition. Hence a literal is of the form $A$ (positive literal) or of the form $\neg A$ (negative literal) where $A$ is an atomic proposition. We denote literals by $L, L_1, \ldots$.

### 3.1.2 Definition (Clause)

A *clause* is a disjunction of literals, that is, it is a formula of the form

$$L_1 \vee \ldots \vee L_n$$

where each $L_i$ is a literal. We denote clauses by $C, C_1, \ldots$. In this definition $n = 1$ is permitted. Hence every literal is also a clause.

### 3.1.3 Definition (Conjunctive Normal Form)

A formula is in *conjunctive normal form*, abbreviated *CNF*, if it is of the form

$$C_1 \wedge \ldots \wedge C_n$$

where each $C_i$ is a clause. The case $n = 1$ is permitted. Therefore, every clause is also a CNF.

### 3.1.4 Examples

Let $A, B, C$ be atomic propositions.

The formulas $A$ and $\neg A$ are literals.

The formula $\neg A \vee \neg B \vee C$ is a clause.

The formula $A \wedge (\neg A \vee B \vee C) \wedge (B \vee \neg C)$ is a CNF.

### 3.1.5 Exercise

Let $A, B, C$ be atomic propositions. Which of the following formulas are CNFs?

(a) $\neg A$

(b) $B \vee \neg C$

(c) $B \wedge \neg C$

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

(e) $\neg(A \vee B)$

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

### 3.1.6 Exercise (pigeon hole principle)

We assume that $A_1, A_2, B_1, B_2, C_1, C_2$ are atomic propositions. Consider the following CNFs:

$$(A_1 \vee A_2) \wedge (B_1 \vee B_2) \wedge (\neg A_1 \vee \neg B_1) \wedge (\neg A_2 \vee \neg B_2)$$

$$\begin{array}{l}(A_1 \vee A_2) \wedge (B_1 \vee B_2) \wedge (C_1 \vee C_2) \wedge \\ (\neg A_1 \vee \neg B_1) \wedge (\neg A_1 \vee \neg C_1) \wedge (\neg B_1 \vee \neg C_1) \wedge \\ (\neg A_2 \vee \neg B_2) \wedge (\neg A_2 \vee \neg C_2) \wedge (\neg B_2 \vee \neg C_2)\end{array}$$

Which is satisfiable?

Hint: Think of of $A_i$ as expressing that pigeon $A$ sits in hole $i$, similarly for $B_i$, $C_i$. Then the first formula expresses that each of the two pigeons $A, B$ sits in one of the two holes, but no two pigeons sit in the same hole.

Similarly, the second formula expresses that the three pigeons $A, B, C$ are sitting in two holes with no two pigeons sitting in the same hole.

### 3.1.7 Exercise

Write down the pigeon hole formula for 2 pigeons and 3 holes.

## 3.2 Computational complexity of satisfiability for CNFs

A formula is said to be in *$n$-CNF*, where $n$ is a natural number, if it is in CNF and every clause contains at most $n$ literals. For example the pigeon hole formulas above are in 2-CNF. In general, the pigeon hole formula for $n$ pigeons and $m$ holes is in $m$-CNF.

### 3.2.1 Theorem (Stephen Cook, 1971)

**SAT**, the satisfiability problem for formulas in CNF, is **NP**-complete (S. A. Cook, *The Complexity of Theorem-Proving Procedures*, Proc. 3rd ACM Symp. Theory of Computing, May 1971).

**NP**-completeness still hold for **3-SAT**, the satisfiability problem restricted to formulas in 3-CNF.

On the other hand, **2-SAT**, the satisfiability problem for formulas in 2-CNF, is solvable in polynomial time (for those interested in complexity theory: it is **NL**-complete, that is, complete for all non-deterministic log-space problems).

A good introduction to complexity classes (such as **NP**, **NL** and many others) as well is given on the Wikipedia page *Complexity class*. A standard textbook is *Computational complexity* by C. H. Papadimitriou, Addison-Wesley 1994 (available at the Library).

The satisfiability problem for CNF has countless applications in Computer Science and bears many open problems. For example it is still unknown whether satisfiability of CNFs is decidable in polynomial time. It is widely believed that this is not the case. If it were the case, then the complexity classes **P** and **NP** would coincide (*Millennium Problem* **P** versus **NP**).

## 3.3 Horn Formulas

Another important class of formulas are *Horn Formulas*. These are formulas in CNF where each clause is a *Horn clause*, that is, a clause containing at most one positive literal. Hence, a Horn Clause has the form

$$\neg A_1 \vee \ldots \vee \neg A_n \vee B \quad \text{or} \quad \neg A_1 \vee \ldots \vee \neg A_n$$

Horn Clauses can be written equivalently using conjunction and implication:

$$A_1 \wedge \ldots \wedge A_n \to B \quad \text{or} \quad A_1 \wedge \ldots \wedge A_n \to \bot$$

### 3.3.1 Theorem

**HORNSAT**, the satisfiability problem for Horn Formulas, can be solved in polynomial time, in fact it is **P**-complete.

Horn formulas are the basis of *Logic Programming* a style of programming that is used in Computational Linguistics, Expert Systems, Deductive Data Bases, and Artificial Intelligence and machine learning. The dominant logic programming language is *Prolog* (see, for example the textbook *From logic programming to Prolog* by K. R. Apt, Prentice Hall 1996, available at the library).

## 3.4 Disjunctive Normal Form (DNF)

By swapping conjunction ($\wedge$) and disjunction ($\vee$) one obtains a normal form dual to CNF.

### 3.4.1 Definition (Disjunctive Normal Form)

A *disjunctive normal form*, abbreviated *DNF*, is a formula of the form

$$D_1 \vee \ldots \vee D_n$$

where each $D_i$ is a *conjunctive clause*, that is, a conjunction of literals.

### 3.4.2 Example

Let $A, B, C$ be atomic propositions. The following formula is a DNF:

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

Formulas in DNF are less frequently used. However, they have the interesting property that the satisfiability problem for them is trivial.

### 3.4.3 Exercise (satisfiability for DNF)

Prove that satisfiability for DNF formulas is solvable in linear time.

## 3.5 Expressive completeness

A class of formulas is called *expressively complete*, *complete* for short, if every logic gate can be defined by a formula in that class, that is, for every logic gate $g : \{0,1\}^n \to \{0,1\}$ there exists a formula $F$ in the class such that

$$[\![F]\!](x_1,\dots,x_n) = g(x_1,\dots,x_n)$$

for all $(x_1,\dots,x_n) \in \{0,1\}^n$.

It easy to see that CNF and DNF are *complete* classes of formulas, that is, they can define every logic gate.

### 3.5.1 Theorem (completeness of CNF and DNF)

CNF and DNF are complete, that is, every logic gate can be defined by a formula in CNF and also by a formula in DNF.

**Proof.** We prove the theorem for DNF (for CNF the argument is similar and left as an exercise).

$g : \{0,1\}^n \to \{0,1\}$ be a logic gate. We are looking for a formula $F$ in DNF that defines $g$, that is $[\![F]\!]\alpha = g(\alpha)$ for all assignments $\alpha$. Recall that we identify the set of assignments with the set $\{0,1\}^n$ of $n$-tuples of Boolean values by identifying an assignment $\alpha$ with the tuple $(\alpha(P_1),\dots,\alpha(P_n)) \in \{0,1\}^n$.

Recall that there exist $2^n$ assignments. Let $\alpha_1,\dots,\alpha_k$ a listing of those assignments for which $g$ returns 1. That is, $g(\alpha_1) = \dots = g(\alpha_k) = 1$, and $g(\alpha) = 0$ for all other assignments $\alpha$.

We encode each $\alpha_i$ by a conjunctive clause $D_i$ as follows:

$$D_i := L_{i,1} \wedge \dots \wedge L_{i,n}$$

where $L_{i,j} := P_j$ if $\alpha_i(P_j) = 1$ and $L_{i,j} := \neg P_j$ if $\alpha_i(P_j) = 0$.

For example, if $\alpha_i = (0,1,0)$, then $D_i = \neg P_1 \wedge P_2 \wedge \neg P_3$.

Clearly, we have for any assignment $\alpha$ and any $j \in \{1,\dots,n\}$

$$[\![L_{i,j}]\!]\alpha = 1 \quad \Leftrightarrow \quad \alpha(P_j) = \alpha_i(P_j)$$

Therefore,

$$\begin{aligned} [\![D_i]\!]\alpha = 1 \;&\Leftrightarrow\; \text{for all } j \in \{1,\dots,n\}\ [\![L_{i,j}]\!]\alpha = 1 \\ &\Leftrightarrow\; \text{for all } j \in \{1,\dots,n\}\ \alpha(P_j) = \alpha_i(P_j) \\ &\Leftrightarrow\; \alpha = \alpha_i \end{aligned}$$

which means that $D_i$ does indeed properly encode $\alpha_i$.

Now we define the formula we are looking for by

$$F := D_1 \vee \ldots \vee D_k$$

Clearly, $F$ is in DNF. We show that $F$ defines the logic gate $g$. Let $\alpha$ be an assignment. We have to show $[\![F]\!]\alpha = g(\alpha)$, that is, $[\![F]\!]\alpha = 1 \Leftrightarrow g(\alpha) = 1$.

$$\begin{array}{rcl} [\![F]\!]\alpha = 1 & \Leftrightarrow & \text{for some } i \in \{1,\ldots,k\}\ [\![D_i]\!]\alpha = 1 \\ & \Leftrightarrow & \text{for some } i \in \{1,\ldots,k\}\ \alpha = \alpha_i \\ & \Leftrightarrow & g(\alpha) = 1 \end{array}$$

### 3.5.2 Example

Consider the logic gate $g$ of three variables defined by the truth table

| $x$ | $y$ | $z$ | $g(x,y,z)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

There are two assignments where $g$ returns 1: $\alpha_1 := (0,0,1)$ and $\alpha_2 := (1,0,0)$. Hence $D_1 = \neg P_1 \wedge \neg P_2 \wedge P_3$ and $D_2 = P_1 \wedge \neg P_2 \wedge \neg P_3$, and the formula $F$ in DNF which defines $g$ is

$$(\neg P_1 \wedge \neg P_2 \wedge P_3) \vee (P_1 \wedge \neg P_2 \wedge \neg P_3)$$

We call a set of logical connectives *complete* if the class of all formulas defined from them is complete. In this definition we regard the logical constants $\bot$ and $\top$ as connectives.

For example, since DNF is complete, and formulas in DNF only use the connectives $\wedge, \vee, \neg$, it follows, that the set $\{\wedge, \vee, \neg\}$ is complete.

### 3.5.3 Theorem (Complete sets of connectives)

Each of the following sets of connectives is complete:

(a) $\{\wedge, \neg\}$

(b) $\{\vee, \neg\}$

(c) $\{\rightarrow, \bot\}$

**Proof.** (a) Since $\{\wedge, \vee, \neg\}$ is complete it suffices to show that $\vee$ can be defined by $\wedge$ and $\neg$.

Recall that $\neg\neg A \equiv A$ and $\neg(A \vee B) \equiv \neg A \wedge \neg B$. It follows

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

The proof of part (b) is similar and left as an exercise.

To prove (c), it suffices, by (b), to show that $\neg$ and $\vee$ can be defined by $\rightarrow$ and $\bot$. We have $\neg A \equiv A \rightarrow \bot$. Furthermore, since $A \rightarrow B \equiv \neg A \vee B$,

$$A \vee B \equiv \neg\neg A \vee B \equiv \neg A \rightarrow B$$

### 3.5.4 Exercise (Incomplete sets of connectives)

Show that none of the following sets of connectives is complete:

(a) $\{\wedge, \vee\}$

(b) $\{\wedge, \rightarrow\}$

(c) $\{\wedge, \bot\}$

Hint: For (a) and (b) show that the logic gate which is constant 0 is not definable. For (c) show that every formula $F$ definable by $\wedge$ and $\bot$ defines a monotone logic gate, that is, if $\alpha \leq \alpha'$, then $[\![F]\!]\alpha \leq [\![F]\!]\alpha'$ where the order on assignments is defined pointwise. Use induction on formulas.

### 3.5.5 Exercise (Incompleteness of Horn Formulas)

Show that the class of Horn Formulas is not complete.

Hint: Show that if $F$ is a Horn Formula and $\alpha_1$ and $\alpha_2$ are models of $F$ (that is, $[\![F]\!]\alpha_1 = [\![F]\!]\alpha_1 = 1$), then the assignment $\alpha$, defined by $\alpha(A) := \wedge(\alpha_1(A), \alpha_2(A))$, is also a model of $F$. Conclude from this that the logic gate $\vee$ cannot be defined by a Horn Formula.

*Remark.* The hint above implies in particular that if a Horn Formula is satisfiable, then it has a smallest model. This fact is fundamental for Logic Programming since it implies that the standard proof strategy applied in Prolog is complete in the sense that it can derive any facts that are a logical consequence of the given logic program. The notions of "logical consequence" and "proof" will be explained in detail later.

Since, as we have seen in Exercise 3.4.3, the satisfiability problem for DNF is solvable in linear time, it seems that the satisfiability problem for CNF can be efficiently solved by translating CNF into DNF. That this translation is possible, follows from the completeness of DNF. However, the obvious translation derived below, which is derived from the de Morgan Law

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

leads to an *exponential blow up*. Hence, in terms of the satisfiability problem for CNF nothing is gained.

### 3.5.6 Definition (Translation of CNF into DNF)

Let $F = C_1 \wedge \ldots \wedge C_n$ a formula in CNF, that is, each $C_i$ is a disjunctive clause. To simplify matters we assume that all $C_i$ contain the same number $m$ of literals (this can always be achieved by repeating literals). Hence

$$C_i = L_{i,1} \vee \ldots \vee L_{i,m}$$

Then the following formula $G$ in DNF is equivalent to $F$:

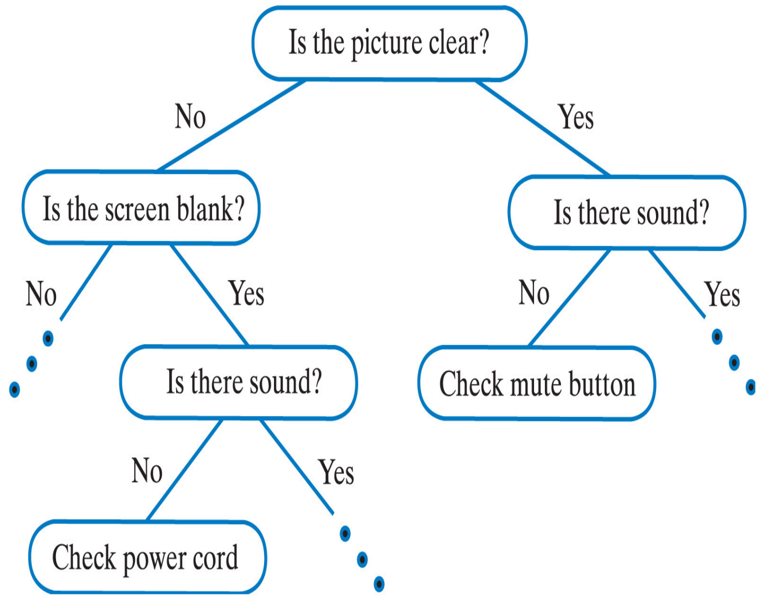
$$G := \bigvee_{\alpha:\{1,\ldots,n\}\to\{1,\ldots,m\}} (L_{1,\alpha(1)} \wedge \ldots \wedge L_{n,\alpha(n)})$$

Since there are $m^n$ functions $\alpha : \{1, \ldots, n\} \to \{1, \ldots, m\}$, we clearly have

$$\mathrm{size}(G) = O\left(2^{\mathrm{size(F)}}\right)$$

## 3.6 Binary Decision Diagrams

A *decision tree* is a tree where each internal node contains a question and the children of a node are the answers to the question.



A fragment of a decision tree (courtesy Univ. Tübingen)

Decision trees can be represented as special formulas using a ternary logical connective $A \to (B, C)$, to be read as "if $A$ then $B$ else $C$". It can easily be defined using implication, conjunction and negation:

$$A \to (B, C) := (A \to B) \wedge (\neg A \to C)$$

The decision tree shown in the image above can now be written as

$$\text{picClear} \to (\text{sound} \to (\ldots, \text{checkMB}), \text{screenBlank} \to (\text{sound} \to (\text{checkPC}, \ldots), \ldots))$$

The truth table of the if-then-else connective $\cdot \to (\cdot, \cdot)$ is as follows

| $x$ | $y$ | $z$ | $x \to (y, z)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

### 3.6.1 Exercise

Let $A, B, C$ be atomic propositions.

(a) Find a formula in CNF that is equivalent to $A \to (B, C)$.

(b) Show that the DNF $(A \land B) \lor (\neg A \land C)$ is equivalent to $A \to (B, C)$.

### 3.6.2 Definition

The set of formulas in *if-then-else normal form* (INF) is inductively defined as follows:

(i) $\bot$ and $\top$ are in INF.

(ii) if $A$ is an atomic proposition and $F$, $G$ are in INF, then $A \to (F, G)$ is in INF.

For example, if $A, B$ are atomic proposition, then the formula

$$F := A \to (B \to (\top, \bot), B \to (\bot, \bot))$$

is in INF.

### 3.6.3 Exercise

(a) Draw the tree for the formula $F$ above.

(b) Find a formula in INF that is equivalent to $F$ but shorter.

(c) Find a formula that is equivalent to $F$ but uses only one basic logical connective

Obviously, every logic gate can be defined by a decision tree and hence by a formula in INF. The decision tree is not uniquely determined: it depends on the order of the variables.

Decision trees given by formulas in INF may contain redundancies:

(1) Repeated variables (atomic propositions) in one branch: these can be eliminated by replacing the subtree beginning at the lower occurrence by its left respectively right subtree if that lower occurrence is in the left respectively subtree of the upper occurrence of that variable.

(2) A node with two identical subtrees: Replace the node by that subtree.

(3) Identical subtrees that don't have a common parent node: these can be avoided by passing from a tree structure to graph structure, more precisely a *directed acyclic graph (DAG)*.

After carrying out these simplifications one arrives at a *binary decision diagram (BDD)*. If in addition one requires that on all paths in the dag the variables appear in a fixed order one has a *reduced ordered BDD (ROBDD)*. However, in the literature ROBDDs are often referred to simply as BDDs.

### 3.6.4 Example

Consider the following decision tree:

$$\begin{tikzcd}
& & & & & & & A \arrow[dllll, "0"'] \arrow[drrrr, "1"] & & & & & & & \\
& & & B \arrow[dll, "0"'] \arrow[drr, "1"] & & & & & & & & B \arrow[dll, "0"'] \arrow[drr, "1"] & & & \\
& C \arrow[dl, "0"'] \arrow[dr, "1"] & & & & C \arrow[dl, "0"'] \arrow[dr, "1"] & & & & C \arrow[dl, "0"'] \arrow[dr, "1"] & & & & C \arrow[dl, "0"'] \arrow[dr, "1"] & \\
\boxed{1} & & \boxed{1} & & \boxed{1} & & \boxed{0} & & \boxed{1} & & \boxed{0} & & \boxed{0} & & \boxed{0}
\end{tikzcd}$$

Simplification (1) does not apply since no variable occurs twice on the same path.

The leftmost subtree beginning with $C$ has only 1s at its leaves. Hence, applying simplification (2), it may be replaced by the leaf 1. Similarly, the rightmost subtree beginning with $C$ has only 0s at its leaves. Therefore we replace it by the leaf 0.

Carrying out these simplifications we obtain:

$$\begin{tikzcd}
 & & & & A \arrow[dll, "0"'] \arrow[drr, "1"] & & & \\
 & & B \arrow[dl, "0"'] \arrow[dr, "1"] & & & & B \arrow[dl, "0"'] \arrow[dr, "1"] & \\
 & \boxed{1} & & C \arrow[dl, "0"'] \arrow[d, "1"] & & C \arrow[d, "0"'] \arrow[dr, "1"] & & \boxed{0} \\
 & & \boxed{1} & \boxed{0} & & \boxed{1} & \boxed{0} &
\end{tikzcd}$$

Since the two remaining subtrees starting with $C$ are identical we my identify them:

$$\begin{tikzcd}
 & & A \arrow[dl, "0"'] \arrow[dr, "1"] & & \\
 & B \arrow[dl, "0"'] \arrow[dr, "1"] & & B \arrow[dl, "0"'] \arrow[dr, "1"] & \\
\boxed{1} & & C \arrow[dl, "0"'] \arrow[dr, "1"] & & \boxed{0} \\
 & \boxed{1} & & \boxed{0} &
\end{tikzcd}$$

Finally, we identify all leaves with the same result:

$$\begin{tikzcd}
 & & A \arrow[dl, "0"'] \arrow[dr, "1"] & & \\
 & B \arrow[ddl, "0"'] \arrow[dr, "1"] & & B \arrow[dl, "0"'] \arrow[ddr, "1"] & \\
 & & C \arrow[dll, "0"'] \arrow[drr, "1"] & & \\
\boxed{1} & & & & \boxed{0}
\end{tikzcd}$$

Most applications of BDDs rely on the fact that every logic gate is implemented by exactly one ROBDD when the order of variables is fixed:

### 3.6.5 Canonicity Theorem for BDDs

For every logic gate $g$ of $n$ variables and every order of the $n$ input variables there exists exactly one reduced ordered BDD defining $g$.

### 3.6.6 Choosing the variable ordering

The ordering of variables in a BDD is crucial. There are logic gates whose sizes of ROBDDs vary between linear and exponential, depending on the chosen order of variables. For example, the ROBDD of the DNF

$$(A_1 \wedge A_2) \vee (A_3 \wedge A_4) \vee \ldots \vee (A_{2n-1} \wedge A_{2n})$$

has $2n + 2$ nodes when the variable order $A_1, A_2, A_3, A_4 \ldots, A_{2n-1}, A_{2n}$ is chosen. However, for the order $A_1, A_3, \ldots, A_{2n-1}, A_2, A_4, \ldots, A_{2n}$ the ROBDD has $2^{n+1}$ nodes.

Finding the optimal variable ordering is a hard problem. More precisely, this problem is **NP**-hard (Sieling, 2002). Deciding whether a given order is optimal is **NP**-complete (Bollig/Wegener, 1996).

### 3.6.7 Applications of BDDs

BDDs are heavily used in CAD software, equivalence and correctness testing of digital circuits, synthesizing circuits and formal verification, in particular model checking.

BDDs were systematically studied first by *Randal E. Bryant (Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 8(C-35), 1986).*

There are several BDD packages that are able to efficiently construct and manipulate BDDs, for example BuDDy (Kopenhagen), CUDD (Boulder).

### 3.6.8 Exercises

(a) Draw the truth table of the logic gate defined by the BDD (or decision tree) of Example 3.6.4.

(b) Find a DNF defining the logic gate in (a).

(c) Find a CNF defining the logic gate in (a).

(d) Construct the decision tree for the logic gate in (a) with respect to the variable ordering $C, A, B$.

(e) Transform the decision tree in (d) into an ROBDD.

(f) Construct ROBDDs for the formula

$$(A_1 \wedge A_2) \vee (A_3 \wedge A_4) \vee \vee(A_3 \wedge A_4)$$

with respect to the following variable orderings (see the discussion in Section 3.6.6):

(i) $A_1, A_2, A_3, A_4, A_5, A_6$

(ii) $A_1, A_3, A_5, A_2, A_4, A_6$

(g) Show that a program that transforms a CNF into the corresponding ROBDD (with respect of some given variable ordering) can be used to solve the satisfiability problem for CNFs.

(h) What can you deduce from (g) regarding the computational complexity of computing a reduced ordered BDD from a CNF (without using other complexity results about ROBDDs)?

(i) Give the ROBDD for the formula given in Exercise 3.1.6 expressing the pigeon hole principle for three pigeons and two holes.

(j) Write down a formula that expresses the pigeon hole principle for two pigeons and two holes.

(k) Is the formula in (j) a Horn formula?

(l) Draw the truth table for the logic gate defined by the formula in (j).

(m) Give the ROBDD for the logic gate you found in (l).

# 4 The Compactness Theorem

The Compactness Theorem is a fundamental result in logic with many applications in mathematics, automated theorem proving, satisfiability testing and other areas. The theorem refers to the notion of satisfiability of a (typically infinite) set of formulas. We first define this and related notions.

## 4.1 Satisfiability and logical consequence for sets of formulas

Since we consider infinite sets of formulas we also need to consider infinite assignments which assign to infinitely many atomic propositions $P_1$, $P_2$, ... truth values in $\{0, 1\}$. Hence an assignment can either be

finite (as before), that is $\alpha = (\alpha(P_1), \alpha(P_2), \dots, \alpha(P_n)) \in \{0, 1\}^n$,

or infinite, that is $\alpha = (\alpha(P_1), \alpha(P_2), \dots) \in \{0, 1\}^{\mathbf{N}}$.

### 4.1.1 Definition (Model, Satisfiability, for a set of formulas)

Let $\Gamma$ be a set of formulas.

An assignment $\alpha$ is a *model* of $\Gamma$ if $[\![F]\!]\alpha = 1$ for all formulas $F \in \Gamma$. One also says $\alpha$ *satisfies* $\Gamma$ and writes

$$\alpha \models \Gamma$$

$\Gamma$ is *satisfiable* if it has a model.

If a set of formulas has no model, then it is called *unsatisfiable*.

**Warning:** It is *not* true that a set of formulas $\Gamma$ is satisfiable if and only if all formulas in $\Gamma$ are satisfiable! Give a counterexample!

### 4.1.2 Definition (Logical consequence)

A formula $G$ is a *logical consequence* of a set of formulas $\Gamma$ if every model of $\Gamma$ is a model of $G$, that is, for all assignments $\alpha$, if $\alpha \models \Gamma$, then $\alpha \models G$ (recall that $\alpha \models G$ means $[\![G]\!]\alpha = 1$). One also says $\Gamma$ *logically implies* $G$ and writes

$$\Gamma \models G$$

*Remarks*.

1. A formula $G$ is a tautology if and only $\emptyset \models G$. Instead of $\emptyset \models G$ one also writes just $\models G$.

2. For finite sets of formulas the new notions can be explained in terms of notions introduced earlier:

$$\{F_1, \ldots, F_n\} \text{ is satisfiable if and only } F_1 \land \ldots \land F_n \text{ is satisfiable,}$$

$$\{F_1, \ldots, F_n\} \models G \text{ if and only } F_1 \land \ldots \land F_n \to G \text{ is a tautology.}$$

### 4.1.3 Exercise

Let $\Gamma$ be a set of formulas and $G$ a formula. Show:

(a) $\Gamma \models G$ if and only if $\Gamma \cup \{\neg G\}$ is unsatisfiable.

(b) If $\Gamma \models G$ and $\Gamma$ is satisfiable, then $G$ is satisfiable.

**Solution.**

(a) For the implication from left to right assume $\Gamma \models G$. We show that $\Gamma \cup \{\neg G\}$ is unsatisfiable. Let $\alpha$ be an assignment. We have to show $\alpha \not\models \Gamma \cup \{\neg G\}$.

Case 1: $\alpha \not\models \Gamma$. Then $\alpha \not\models \Gamma \cup \{\neg G\}$.

Case 2: $\alpha \models \Gamma$. Since we assumed that $\Gamma \models G$, it follows that $\alpha \models G$, that is $[\![G]\!]\alpha = 1$. Hence $[\![\neg G]\!]\alpha = 0$, that is, $\alpha \not\models \neg G$. Therefore $\alpha \not\models \Gamma \cup \{\neg G\}$.

For the opposite implication we assume that $\Gamma \cup \{\neg G\}$ is unsatisfiable. We show $\Gamma \models G$. Assume $\alpha \models \Gamma$. We have to show $\alpha \models G$. Assume not, that is $[\![G]\!]\alpha = 0$. Then $[\![\neg G]\!]\alpha = 1$, that is $\alpha \models \neg G$. Hence $\alpha \models \Gamma \cup \{\neg G\}$ contradicting the assumption that $\Gamma \cup \{\neg G\}$ is unsatisfiable.

(b) Assume that $\Gamma \models G$ and $\Gamma$ is satisfiable. We show that $G$ is satisfiable. Since $\Gamma$ is satisfiable, there exists an assignment $\alpha$ such that $\alpha \models \Gamma$. Since we assumed that $\Gamma \models G$, it follows that $\alpha \models G$. Hence $G$ is satisfiable.

### 4.1.4 Exercises (Schoening, Ex. 7)

Give an example of a set $\Gamma$ of 3 formulas such that $\Gamma$ is unsatisfiable, but every 2-element subset of it is satisfiable.

**Solution.** Consider the 3 formulas $A$, $B$ and $\neg A \lor \neg B$ where $A$ and $B$ are atomic. It is easy to check that these have the desired property.

## 4.2 König's Lemma

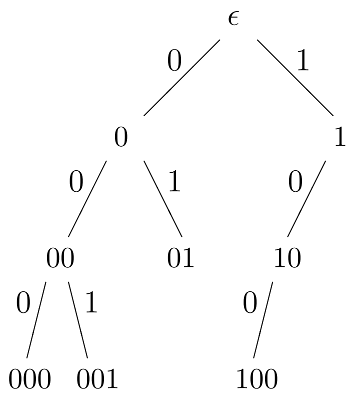
The essence of the proof of the Compactness Theorem is a theorem about binary trees called *König's Lemma*.

### 4.2.1 Definition (Binary Tree)

Let $\beta \in \{0,1\}^n$ and $\beta' \in \{0,1\}^m$ be finite assignments. We say $\beta'$ is a *restriction* of $\beta$, or $\beta$ is an *extension* of $\beta'$, written $\beta' \subseteq \beta$, if $m \le n$ and $\beta'(P_i) = \beta(P_i)$ for all $i \in \{1, \ldots, m\}$.

A *binary tree* is a set $T$ of finite assignments which is closed under restriction, that is, if $\beta \in T$ and $\beta' \subseteq \alpha$, then $\beta' \in T$. The elements of $T$ are also called the *nodes* of $T$. The length of a node is also called its *depth*.

An *infinite path* in a binary tree $T$ is an infinite assignment $\alpha = (\alpha(P_1), \alpha(P_2), \ldots)$ such that for all $n \in \mathbf{N}$ the finite assignment $\overline{\alpha}(n) := (\alpha(P_1), \ldots, \alpha(P_n))$ is in $T$.



The binary tree $\{\epsilon, 0, 1, 00, 01, 10, 000, 001, 100\}$

### 4.2.2 Theorem (König's Lemma)

Every infinite binary tree has an infinite path.

**Proof.** Let $T$ be an infinite binary tree.

We define an infinite assignment $\alpha = (\alpha(P_1), \alpha(P_2), \ldots)$ as follows:

We set $\alpha(P_1) := 0$ if for infinitely many $\beta \in T$ we have $\beta(P_1) = 0$ (that is, the left subtree of $T$ is infinite), otherwise we set $\alpha(P_1) := 1$ (in that case the right subtree of $T$ is infinite, because $T$ is infinite).

In any case there are infinitely many assignments $\beta \in T$ with $\beta(P_1) = \alpha(P_1)$.

With a similar argument we define $\alpha(P_2) \in \{0, 1\}$ such that there are infinitely many assignments $\beta \in T$ with $\beta(P_1) = \alpha(P_1)$ and $\beta(P_2) = \alpha(P_2)$.

Repeating this procedure over and over again we obtain an infinite assignment $\alpha = (\alpha(P_1), \alpha(P_2), \ldots)$ such that for each $n$ there are infinitely many $\beta \in T$ with $\beta(P_i) = \alpha(P_i)$ for all $i \in \{1, \ldots, n\}$. Since $T$ is closed under restrictions it follows in particular that $\overline{\alpha}(n) \in T$ for all $n \in \mathbf{N}$, that is, $\alpha$ is an infinite path in $T$. This completes the proof.

*R*emarks.

1. The construction of the infinite path $\alpha$ is not effective since it is impossible to decide effectively which subtree of an infinite binary tree is infinite. In fact, there exists a computable infinite tree, the so-called *Kleene tree*, which has no infinite computable path (it has, of course an infinite path, according to König's Lemma, but this path is not computable).

2. König's Lemma holds more generally for finitely branching trees (instead of just binary trees), that is, trees where every node has only finitely many children. König's Lemma fails for infinitely branching trees, even if we strengthen the hypothesis of $T$ being infinite to $T$ being infinitely deep, that is, having arbitrarily deep nodes.

## 4.3 Statement and proof of the Compactness Theorem

With the help of König's lemma it is now easy to prove the Compactness Theorem.

### 4.3.1 Theorem (Compactness Theorem)

Let $\Gamma$ be a set of formulas such that every finite subset of $\Gamma$ is satisfiable. Then $\Gamma$ is satisfiable.

**Proof.** If there are only finitely many atomic formulas occurring in $\Gamma$, then the number of logic gates defined by some formula in $\Gamma$ is finite (though very large). Therefore, we can select finitely many formulas $F_1, \ldots, F_n$ such that any such logic gate is defined by some $F_i$. Hence every $F \in \Gamma$ is equivalent to some $F_i$. By assumption, the set $\{F_1, \ldots, F_n\}$ has a model $\alpha$ which necessarily is then a model of $\Gamma$.

Hence, we consider now the interesting case that $\Gamma$ is infinite, and there are infinitely many atomic propositions occurring in $\Gamma$, say

$$P_1, P_2, \ldots$$

For every natural number $n$ we let $\Gamma_n$ be the set of those formulas in $\Gamma$ which contain at most the atomic propositions $P_1, \ldots, P_n$. Clearly

$$\Gamma = \bigcup_{n \in \mathbf{N}} \Gamma_n$$

By the argument above, each $\Gamma_n$ is satisfiable (since there are only finitely many atomic formulas occurring in $\Gamma_n$).

We set

$$T_n := \{\beta \in \{0,1\}^n \mid \beta \models \Gamma_n\}$$

Hence, $T_n$ is the set of those finite assignments $\beta$ that assign values to $P_1, \ldots, P_n$ (and maybe other atomic propositions) and satisfy $\Gamma_n$.

Since $\Gamma_n$ is satisfiable, $T_n$ is non-empty for each $n$.

Let $T$ be the union of all the $T_n$. Clearly, $T$ is a tree, since if $\beta = (b_1, \ldots, b_n) \in T_n$ and $\beta' \subseteq \beta$, say $\beta' = (b_1, \ldots, b_k)$ with $k \leq n$, then $\beta' \in T_k$.

Since all the $T_n$ are nonempty, $T$ is infinite. By König's Lemma, $T$ has an infinite path

$$\alpha = (a_1, a_2, \ldots)$$

This means that $\alpha(P_1) = a_1$, $\alpha(P_2) = a_2$, and so on, and $\overline{\alpha}(n) = (a_1, \ldots, a_n) \in T_n$. Hence $\overline{\alpha}(n) \models \Gamma_n$ and consequently $\alpha \models \Gamma_n$ for all $n$. Therefore, $\alpha \models \Gamma$. This completes the proof.

### 4.3.2 Corollary

(a) If a set of formulas is unsatisfiable, then it has a finite unsatisfiable subset.

(b) If $\Gamma \models G$, then $\Gamma_0 \models G$ for some finite subset $\Gamma_0$ of $\Gamma$.

**Proof.**

(a) This is the contrapositive of the Compactness Theorem.

(b) Assume $\Gamma \models G$. By Exercise , $\Gamma \cup \{\neg G\}$ is unsatisfiable. By part (a) above, there exists a finite unsatisfiable subset $\Gamma_1$ of $\Gamma \cup \{\neg G\}$. Set $\Gamma_0 := \Gamma_1 \cap \Gamma$. Clearly, $\Gamma_0$ is finite. Furthermore, $\Gamma_1 \subseteq \Gamma_0 \cup \{\neg G\}$. Hence $\Gamma_0 \cup \{\neg G\}$ is unsatisfiable. By Exercise it follows that $\Gamma_0 \models G$.

### 4.3.3 Definition (Axiom system)

Let $\Gamma$ be a set of formulas. A set of formulas $\Gamma_0$ is an *axiom system* for $\Gamma$ if $\Gamma$ and $\Gamma_0$ have the same models, that is for every assignment $\alpha$,

$$\alpha \models \Gamma \quad \text{if and only if} \quad \alpha \models \Gamma_0$$

A set of formulas is called *finitely axiomatisable* if it has a finite axiom system.

### 4.3.4 Exercise

(a) Show that $\Gamma_0$ is an axiom system for $\Gamma$ if and only if

  (i) $\Gamma \models G$ for every $G \in \Gamma_0$ and

  (ii) $\Gamma_0 \models F$ for every $F \in \Gamma$.

(b) Show that $\Gamma$ is finitely axiomatisable if and only if there exists a formula $G$ such that

  (i) $\Gamma \models G$ and

  (ii) $\models G \rightarrow F$ (that is, $\{G\} \models F$) for every $F \in \Gamma$.

**Solution.** Part (a) is obvious: (i) is equivalent to the implication "if $\alpha \models \Gamma$ then $\alpha \models \Gamma_0$" (for all $\alpha$), and (ii) is equivalent to the converse implication "if $\alpha \models \Gamma_0$ then $\alpha \models \Gamma$".

Part (b) follows from part (a) since the finite set of formulas $\Gamma_0$, say, $\Gamma_0 = \{F_1, \ldots, F_n\}$ can be replaced by the conjunction of its elements $F_1 \wedge \ldots \wedge F_n$.

### 4.3.5 Exercise

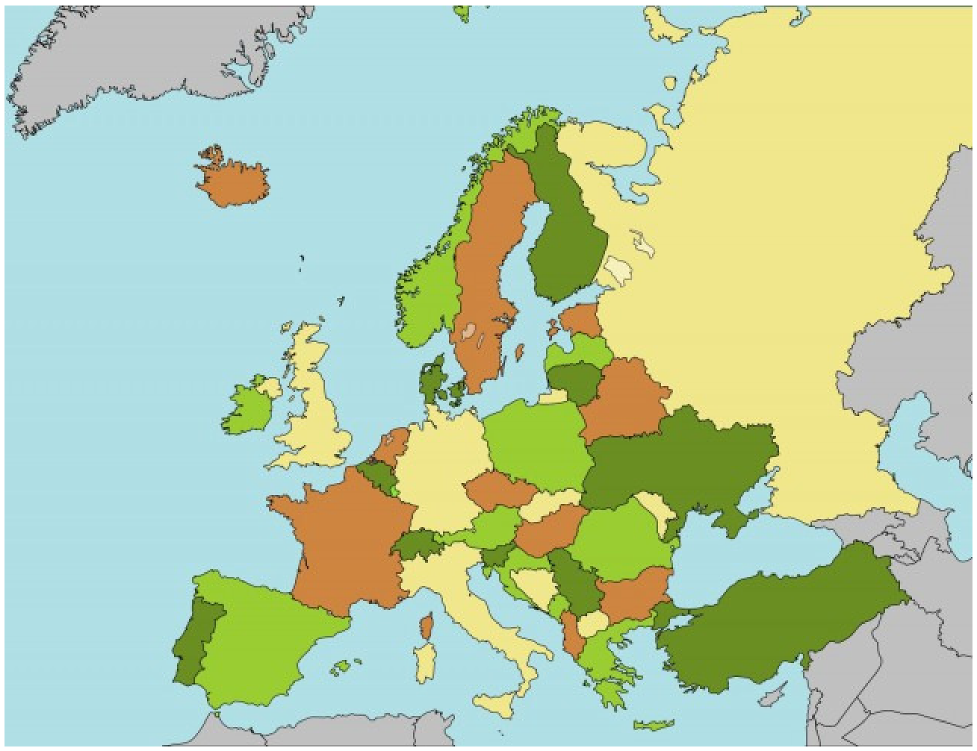
Let $\Gamma = \{F_1, F_2, \ldots\}$ be an infinite set of formulas such that $\models F_{n+1} \rightarrow F_n$, but $\not\models F_n \rightarrow F_{n+1}$ for all $n \in \mathbf{N}$. Show that $\Gamma$ is not finitely axiomatisable.

**Solution.** If $\Gamma$ were finitely axiomatisable, then by Exercise 4.3.4 (b), there would exist a formula $G$ such that $\Gamma \models G$ and $\models G \rightarrow F$ for every $F \in \Gamma$. By the Compactness Theorem, there exists a finite subset $\Gamma_0$ of $\Gamma$ such that $\Gamma_0 \models G$. Since $\Gamma_0$ is finite, there exists $k \in \mathbf{N}$ such that $\Gamma_0 \subseteq \{F_1, \ldots, F_k\}$. Since $\models F_{n+1} \rightarrow F_n$ for all $n \in \mathbf{N}$, it follows that $\models F_k \rightarrow F$ for all $F \in \Gamma_0$. Since $\Gamma_0 \models G$, it follows that $F_k \models G$. Since $\models G \rightarrow F$ for all $F \in \Gamma$, we have in particular that $\models G \rightarrow F_{k+1}$. Therefore, $\models F_k \rightarrow F_{k+1}$, contradicting the assumption that $\not\models F_n \rightarrow F_{n+1}$ for all $n \in \mathbf{N}$.

### 4.3.6 Example (Four Colour Theorem)

As an application of the Compactness Theorem we show that the Four Colour Theorem for infinite maps or graphs can be derived from its finite version. Below we mean by a *map* a separation of the plane into regions, called countries. A map is finite if the number of regions is finite.

**Four Colour Theorem.** Every finite map can be coloured by 4 colours such that neighbouring countries have different colours.



A map of Europe coloured using only four colours (Wikimedia Commons)

The theorem was proven in 1975 by Appel and Haken with the help of a computer. Since the proof was too complex to be checked by a human, some doubts remained about its validity. In 1997, Robertson, Sanders, Seymour, and Thomas gave a simpler proof which still required help by a computer, and in 2005 the whole proof was formalized by Gonthier in the interactive proof system Coq. Now the proof is widely accepted. The Four Colour Theorem is the first theorem for which no proof without computer assistance is known.

Deciding whether a given finite map can be coloured with only three colours is an **NP**-complete problem (Dayley 1980).

The Four Colour Theorem for finite maps can be restated in terms of undirected planar graphs (a graph is planar if it can be drawn in the plane without crossing edges), where the nodes corresponds to a country and two nodes that are connected by an edge correspond to neighbouring countries:

> Each finite planar graph has a colouring of the nodes (or vertices) using not more than four colours such that nodes connected by an edge have different colours.

The infinite Four Colour Theorem is the same statement, but for infinite maps, that is, infinite planar graphs.

To show that the infinite Four Colour Theorem holds, we express the statement by a set of propositional formulas.

Let $G$ be an infinite planar graph.

For every $n \in \mathrm{N}$ (representing a node or country) and every $c \in \{1, 2, 3, 4\}$ representing a colour) we introduce an atomic proposition $P_{n,c}$ with the intended meaning that node $n$ has colour $c$.

We write down formulas that express that $G$ has a 4-colouring.

1. $P_{n,1} \vee P_{n,2} \vee P_{n,3} \vee P_{n,4}$ for every $n \in \mathrm{N}$ (every node has a colour)
2. $\neg(P_{n,c} \wedge P_{n,d})$ for every $n \in \mathrm{N}$ and all $c, d \in \{1, 2, 3, 4\}$, $c \neq d$ (a node can have only one colour).
3. $\neg(P_{n,c} \wedge P_{m,c})$ for all $n, m \in \mathrm{N}$ that are connected by an edge and $c \in \{1, 2, 3, 4\}$ (neighbouring nodes have different colours).

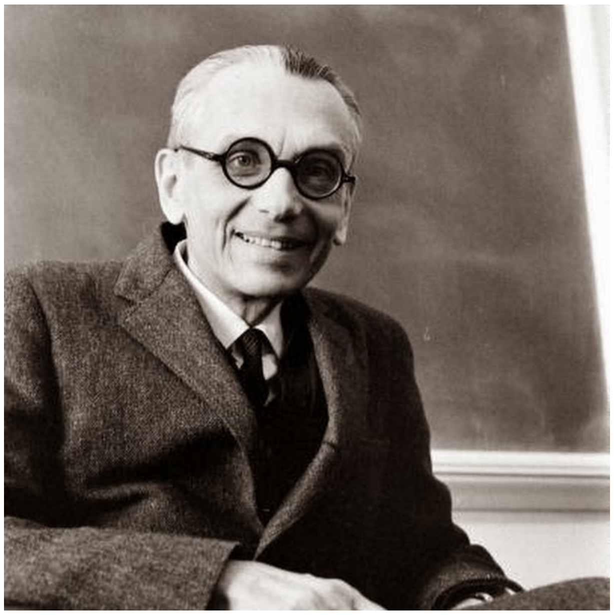
Let $\Gamma$ be the set of all these formulas. Clearly $\Gamma$ is satisfiable if and only if the graph $G$ admits a 4-colouring (and any satisfying assignment will define such a colouring).

By the Compactness Theorem it suffices to show that every finite subset of $\Gamma$ is satisfiable. Let $\Gamma_0$ be such a finite subset. Let $G_0$ the subgraph of $G$ that is formed by those nodes that are mentioned in $\Gamma_0$. $G_0$ is finite since $\Gamma_0$ is finite. By the finite Four Colour Theorem, $G_0$ admits a 4-colouring which defines a satisfying assignment for $\Gamma_0$. Hence $\Gamma_0$ is satisfiable.

This completes the proof.

# 5 Natural Deduction Proofs

We now study a system of simple proof rules for deriving tautologies, that is, logically valid formulas. The famous *Completeness Theorem*, by the Austrian logician *Kurt Gödel*, states that this system of rules suffices to derive in fact *all* tautologies. Gödel proved this theorem for an extension of propositional logic, called *first-order predicate logic*, which we will study later.



Kurt Gödel (1906-1978)

The proof calculus of *Natural Deduction* was first introduced by the German mathematician and logician Gerhard Gentzen (1909-1945) and further developed by the Swedish philosopher and logician Dag Prawitz (born 1936).

Compared with other proof calculi, e.g. Sequent Calculi, or Hilbert Calculi, Natural Deduction has the advantage of being

- close to the natural way of human reasoning, and thus easy to learn;
- closely related to functional programming, and thus is particularly well suited for program synthesis from proofs, which we will study later in the course.

In the following, $A$, $B$, $C$ stand for arbitrary propositional formulas unless stated otherwise. In order to simplify things, from now on we view a negation, $\neg A$, as an abbreviation for the implication $A \to \bot$. This is justified since the two formulas are equivalent.

## 5.1 Definition (Sequent)

A *sequent* is a syntactic object

$$A_1, \ldots, A_k \vdash B$$

where $A_1, \ldots, A_k, B$ are formulas. The idea of a sequent is to express that the assumptions $A_1, \ldots, A_k$ logically imply the conclusion $B$, that is, $\{A_1, \ldots, A_k\} \models B$ holds. $A_1, \ldots, A_k$ are called the *assumptions* (or *antecedent*) and $B$ the *consequence* or *succedent* of the sequent. In the antecedent $A_1, \ldots, A_k$ we ignore the order and multiplicities of formulas, that is, we consider the antecedent as a *set* of formulas. Hence, a more accurate notation for a sequent would be $\{A_1, \ldots, A_k\} \vdash B$, but it is common to omit the curly braces. We often denote the antecedent of a sequent by the letter $\Gamma$. Furthermore, we write $\Gamma, A$ as a shorthand for $\Gamma \cup \{A\}$.

## 5.2 Definition (Derivation)

A *derivation* (or *formal proof*) is a finite tree (drawn correctly, that is, leaves on top and root at the bottom), where each node is labelled by a sequent and a rule according to figure 1 on page 47.

There is an *Assumption rule*, and for each logical connective there are two kinds of rules:

> *Introduction rules*, describing how to *obtain* a formula built from that connective;
>
> *Elimination rules*, describing how to *use* a formula built from that connective.

The sequent at the root of a derivation is called the *end sequent*. The assumption rule could be equivalently written

$$\Gamma \vdash A, \qquad \text{provided } A \in \Gamma$$

The assumptions of the end sequent of a derivation are called the *free assumptions* of a derivation.

## 5.3 Definition (Classical provability)

A sequent $\Gamma \vdash A$ is called *derivable* if there is a derivation with end sequent $\Gamma \vdash A$. We also say that $A$ *is derivable from the assumptions* $\Gamma$.

In the special case when $\Gamma$ is empty, we say that $A$ *is derivable*.

| | Assumption rule $\dfrac{}{\Gamma, A \vdash A}\,\text{use}$ | |
|---|---|---|
| | Introduction rules | Elimination rules |
| $\wedge$ | $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge^+$ | $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge^-_\text{l}$ $\qquad$ $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge^-_\text{r}$ |
| $\to$ | $\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,\to^+$ | $\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\to^-$ |
| $\vee$ | $\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee^+_\text{l}$ $\qquad$ $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee^+_\text{r}$ | $\dfrac{\Gamma \vdash A \vee B \qquad \Gamma \vdash A \to C \qquad \Gamma \vdash B \to C}{\Gamma \vdash C}\,\vee^-$ |
| $\perp$ | | $\dfrac{\Gamma \vdash \perp}{\Gamma \vdash A}\,\text{efq}$ $\qquad$ $\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,\text{raa}$ |

Figure 1: The rules of natural deduction for propositional logic (sequent notation)

## 5.4 Examples

1. We begin with a derivation involving the conjunction introduction rule, $\wedge^+$, and the conjunction elimination rules, $\wedge^-_\text{l}$ and $\wedge^-_\text{r}$.

$$\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\text{use}}{A \wedge B \vdash B}\,\wedge^-_\text{r} \qquad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\text{use}}{A \wedge B \vdash A}\,\wedge^-_\text{l}}{A \wedge B \vdash B \wedge A}\,\wedge^+$$

2. If we add to the derivation in Example 1 an application of the implication introduction rule, $\to^+$, we obtain a proof of $A \wedge B \to B \wedge A$ without assumptions:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\text{use}}{A \wedge B \vdash B}\,\wedge^-_\text{r} \qquad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\text{use}}{A \wedge B \vdash A}\,\wedge^-_\text{l}}{A \wedge B \vdash B \wedge A}\,\wedge^+}{\vdash A \wedge B \to B \wedge A}\,\to^+$$

3. In the following derivation of the formula $A \to (B \to A)$ we use the rule $\to^+$ twice.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{use}
\UnaryInfC{$A, B \vdash A$}
\RightLabel{$\to^+$}
\UnaryInfC{$A \vdash B \to A$}
\RightLabel{$\to^+$}
\UnaryInfC{$\vdash A \to (B \to A)$}
\end{prooftree}
$$

4. Next let us derive $(A \to (B \to C)) \to (A \wedge B \to C)$. Here we use for the first time the implication elimination rule, $\to^-$, also called *modus ponens.* The easiest way to find the derivations is to construct it "bottom up".

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{use}
\UnaryInfC{$A \wedge B \to C, A, B \vdash A \wedge B \to C$}
\AxiomC{}
\RightLabel{use}
\UnaryInfC{$A \wedge B \to C, A, B \vdash A$}
\AxiomC{}
\RightLabel{use}
\UnaryInfC{$A \wedge B \to C, A, B \vdash B$}
\RightLabel{$\wedge^+$}
\BinaryInfC{$A \wedge B \to C, A, B \vdash A \wedge B$}
\RightLabel{$\to^-$}
\BinaryInfC{$A \wedge B \to C, A, B \vdash C$}
\RightLabel{$\to^+$}
\UnaryInfC{$A \wedge B \to C, A \vdash B \to C$}
\RightLabel{$\to^+$}
\UnaryInfC{$A \wedge B \to C \vdash A \to (B \to C)$}
\RightLabel{$\to^+$}
\UnaryInfC{$\vdash (A \wedge B \to C) \to (A \to (B \to C))$}
\end{prooftree}
$$

At the latest at this point we notice that even fairly simple derivations as the one above are very tedious to write down and difficult to read because so many formulas are repeated over and over again in the assumptions of sequents.

The following observation leads to a much more concise notation for derivations that largely avoids repetitions of formulas: The assumptions of a sequent in a derivation consist of formulas that

> are free assumptions, that is, occur in the antecedent of the end sequent of the derivation, or
>
> are "introduced" by the rule $\to^+$.

Furthermore, the only rule where an assumption is "used" is the assumption rule.

Hence, the assumptions may be notationally omitted provided

> the free assumptions are listed with labels, and
>
> the introduction of an assumption via $\to^+$ is indicated by a label.

When an assumption is used in an assumption rule one refers to the corresponding label.

Following this idea, the previous derivation now reads:

$$\dfrac{\dfrac{\dfrac{\dfrac{u : A \wedge B \to C \qquad \dfrac{v : A \qquad w : B}{A \wedge B}\,\wedge^+}{C}\,\to^-}{B \to C}\,\to^+ w}{A \to (B \to C)}\,\to^+ v}{(A \wedge B \to C) \to (A \to (B \to C))}\,\to^+ u$$

Note that the (invisible) assumptions at each point of the derivation consists of the formulas $X$ for which there is a rule of the form

$$\dfrac{Y}{X \to Y}\,\to^+ z$$

below that point. Consequently, assumptions are allowed exactly if they are of the form $z : X$ for such $z$ and $X$. In this example there were no free assumptions.

To display the derivation 1, we list the only free assumption

$u : A \wedge B$

and the derivation is

$$\dfrac{\dfrac{u : A \wedge B}{B}\,\wedge^-_{\mathrm{r}} \qquad \dfrac{u : A \wedge B}{A}\,\wedge^-_{\mathrm{l}}}{B \wedge A}\,\wedge^+$$

## 5.5 Exercise

Write the derivations 2 and 3 above using the new shorthand notation.

The derivation rules themselves can be rewritten in the new style as well, as shown in figure 2 on page 50.

| Assumption rule (*) $u : A$ | | |
|---|---|---|
| | Introduction rules | Elimination rules |
| $\land$ | $\dfrac{A \qquad B}{A \land B}\,\land^+$ | $\dfrac{A \land B}{A}\,\land^-_{\mathrm{l}}$ $\qquad$ $\dfrac{A \land B}{B}\,\land^-_{\mathrm{r}}$ |
| $\to$ | $\dfrac{B}{A \to B}\,\to^+ u$ | $\dfrac{A \to B \qquad A}{B}\,\to^-$ |
| $\lor$ | $\dfrac{A}{A \lor B}\,\lor^+_{\mathrm{l}}$ $\qquad$ $\dfrac{B}{A \lor B}\,\lor^+_{\mathrm{r}}$ | $\dfrac{A \lor B \qquad A \to C \qquad B \to C}{C}\,\lor^-$ |
| $\bot$ | | $\dfrac{\bot}{A}\,\mathrm{efq}$ $\qquad$ $\dfrac{\neg\neg A}{A}\,\mathrm{raa}$ |

Figure 2: The rules of natural deduction for propositional logic (short notation)

The correctness condition (*) for an instance $u : A$ of the assumption rule is that

$u : A$ is among the listed free assumptions, or

below the occurrence of that assumption rule there is a rule of the form

$$\dfrac{B}{A \to B}\,\to^+ u$$

In the following, when writing down concrete derivations, we will always work with the shorthand notation.

5. In order to familiarise ourselves with the *disjunction introduction rules*, $\lor^+_{\mathrm{l}}$, $\lor^+_{\mathrm{r}}$, and the *disjunction elimination rule*, $\lor^-$, we derive the formula $A \lor B \to B \lor A$.

$$\dfrac{\dfrac{u : A \lor B \qquad \dfrac{\dfrac{v : A}{B \lor A}\,\lor^+_{\mathrm{r}}}{A \to B \lor A}\,\to^+ v \qquad \dfrac{\dfrac{w : B}{B \lor A}\,\lor^+_{\mathrm{l}}}{B \to B \lor A}\,\to^+ w}{B \lor A}\,\lor^-}{A \lor B \to B \lor A}\,\to^+ u$$

6. As a slightly more complicated example we derive (one half of) one of de-Morgan's laws, $A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$.

$$
\begin{prooftree}
\AxiomC{$u : A \wedge (B \vee C)$}
\RightLabel{$\wedge^-_{\mathrm{r}}$}
\UnaryInfC{$B \vee C$}
\AxiomC{$u : A \wedge (B \vee C)$}
\RightLabel{$\wedge^-_{\mathrm{l}}$}
\UnaryInfC{$A$}
\AxiomC{$v : B$}
\RightLabel{$\wedge^+$}
\BinaryInfC{$A \wedge B$}
\RightLabel{$\vee^+_{\mathrm{l}}$}
\UnaryInfC{$(A \wedge B) \vee (A \wedge C)$}
\RightLabel{$\to^+ v$}
\UnaryInfC{$B \to (A \wedge B) \vee (A \wedge C)$}
\AxiomC{$u : A \wedge (B \vee C)$}
\RightLabel{$\wedge^-_{\mathrm{l}}$}
\UnaryInfC{$A$}
\AxiomC{$w : C$}
\RightLabel{$\wedge^+$}
\BinaryInfC{$A \wedge C$}
\RightLabel{$\vee^+_{\mathrm{r}}$}
\UnaryInfC{$(A \wedge B) \vee (A \wedge C)$}
\RightLabel{$\to^+ w$}
\UnaryInfC{$C \to (A \wedge B) \vee (A \wedge C)$}
\RightLabel{$\vee^-$}
\TrinaryInfC{$(A \wedge B) \vee (A \wedge C)$}
\RightLabel{$\to^+ u$}
\UnaryInfC{$A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$}
\end{prooftree}
$$

7. Finally, we turn our attention to the rules concerning absurdity, $\perp$, namely *ex-falso-quodlibet*, efq, and *reductio-ad-absurdum*, raa. Recall that $\neg A$ is shorthand for $A \to \perp$, and therefore $\neg\neg A$ stands for $(A \to \perp) \to \perp$. We derive *Peirce's law* $(A \to B) \to A \to A$:

$$
\begin{prooftree}
\AxiomC{$v : \neg A$}
\AxiomC{$u : (A \to B) \to A$}
\AxiomC{$v : \neg A$}
\AxiomC{$w : A$}
\RightLabel{$\to^-$}
\BinaryInfC{$\perp$}
\RightLabel{efq}
\UnaryInfC{$B$}
\RightLabel{$\to^+ w$}
\UnaryInfC{$A \to B$}
\RightLabel{$\to^-$}
\BinaryInfC{$A$}
\RightLabel{$\to^-$}
\BinaryInfC{$\perp$}
\RightLabel{$\to^+ v$}
\UnaryInfC{$\neg\neg A$}
\RightLabel{raa}
\UnaryInfC{$A$}
\RightLabel{$\to^+ u$}
\UnaryInfC{$((A \to B) \to A) \to A$}
\end{prooftree}
$$

8. The rule ex-falso-quodlibet is weaker than reductio-ad-absurdum in the sense that the former can be obtained from the latter: From the assumption $\perp$ we can derive any formula $A$ without using ex-falso-quodlibet (but using reductio-ad-absurdum instead):

$$
\begin{prooftree}
\AxiomC{$\perp$}
\RightLabel{$\to^+ u$}
\UnaryInfC{$(A \to \perp) \to \perp$}
\RightLabel{raa}
\UnaryInfC{$A$}
\end{prooftree}
$$

## 5.6 Definition

Let $\Gamma$ be a (possibly infinite) set of formulas and $A$ a formula.

$\Gamma \vdash_{\mathrm{c}} A \quad :\Leftrightarrow \quad \Gamma_0 \vdash A$ is derivable for some finite subset $\Gamma_0$ of $\Gamma$.

($A$ is derivable from $\Gamma$ in *classical logic*)

$\Gamma \vdash_{\mathrm{i}} A \quad :\Leftrightarrow \quad \Gamma \vdash_{\mathrm{c}} A$ with a derivation not using the rule reductio-ad-absurdum.

($A$ is derivable from $\Gamma$ in *intuitionistic logic*)

$\Gamma \vdash_{\mathrm{m}} A \quad :\Leftrightarrow \quad \Gamma \vdash_{\mathrm{c}} A$ with a derivation using neither the rule reductio-ad-absurdum nor the rule ex-falso-quodlibet.

($A$ is derivable from $\Gamma$ in *minimal logic*)

## 5.7 Soundness and completeness

The soundness and completeness theorems below state that the logical inference rules introduced above precisely capture the notion of logical consequence.

## 5.8 Soundness Theorem

If $\Gamma \vdash_{\mathrm{c}} A$ then $\Gamma \models A$.

**Proof.** The theorem follows immediately from the following statement which can be easily shown by induction on derivations:

For every set of formulas $\Gamma$ and every formula $A$,

if $\Gamma \vdash_{\mathrm{c}} A$ then $\alpha \models A$ for all models $\alpha$ of $\Gamma$

.

Whilst the soundness theorem is not very surprising, because it just states that the inference rules are correct, the following completeness theorem proved by Gödel, states that the logical inference rules above in fact capture all possible ways of correct reasoning.

## 5.9 Completeness Theorem

If $\Gamma \models A$ then $\Gamma \vdash_{\mathrm{c}} A$.

In words: If $A$ is a logical consequence of $\Gamma$ (i.e. $A$ is true in all models of $\Gamma$), then this can be formally derived by the inference rules of natural deduction.

It is important that the Completeness Theorem holds for other logics as well. In particular it holds for *first-order predicate logic* which we will study later.

For propositional logic, the Completeness Theorem is an easy consequence of the Compactness Theorem:

## 5.10 Exercise

Sketch a proof of the Completeness Theorem for Propositional Logic using the Compactness Theorem and the fact that every formula is provably equivalent to a formula in CNF.

**Solution.**

Assume $\Gamma \models A$. By the Compactness Theorem, there are finitely many formulas $B_1, \ldots, B_n \in \Gamma$ such that $\{B_1, \ldots, B_n\} \models A$. Hence $B_1 \wedge \ldots \wedge B_n \to A$ is a tautology.

It suffices to show that $B_1 \wedge \ldots \wedge B_n \to A$ is derivable in classical logic, that is $\vdash_{\mathrm{c}} B_1 \wedge \ldots \wedge B_n \to A$. Because then $\{B_1, \ldots, B_n\} \vdash A$ is derivable and consequently $\Gamma \vdash_{\mathrm{c}} A$.

Therefore, in order to complete the proof of the Completeness Theorem, it suffices to show that all propositional tautologies are derivable in classical logic. This can be done as follows: Any formula $F$ can be transformed into an equivalent formula in Conjunctive Normal Form, that is,

$$F \equiv C_1 \wedge \ldots \wedge C_n$$

where each $C_i$ is a disjunctive clause. Moreover the equivalence can be formally proven, that is,

$$\vdash_{\mathrm{c}} F \leftrightarrow C_1 \wedge \ldots \wedge C_n$$

To construct the derivation one uses formal proofs of the de-Morgan laws and laws expressing implication in terms of disjunction and negation. This is a bit tedious, but easy.

Now, $F$ is a tautology if and only if every $C_i$ is a tautology. Hence, we further reduced the problem to showing that all disjunctive clauses which are tautologies are derivable.

So, take any disjunctive clause

$$C = L_1 \vee \ldots \vee L_k$$

Clearly, $C$ is a tautology if and only if it contains two complementary literals, say $L_i = A$ and $L_j = \neg A$ where $A$ is an atomic proposition. But $A \vee \neg A$ is derivable in classical logic. Hence $C$ is derivable.

The following consequences of the Completeness Theorem refer to the notion of consistency.

## 5.11 Definition (Formal consistency)

A (possibly infinite) set of formulas $\Gamma$ is called **formally consistent** if $\Gamma \nvdash_c \bot$, that is there is no (classical) derivation of $\bot$ from assumptions in $\Gamma$.

In other words: A set of formulas is consistent if and only if no contradiction can be derived from it.

## 5.12 Lemma

A set of formulas $\Gamma$ is satisfiable if and only if $\Gamma \not\models \bot$.

**Proof.** "$\Rightarrow$": Assume that $\Gamma$ is satisfiable. This means that there exists an assignment $\alpha$ such that $\alpha \models \Gamma$. Clearly $\alpha \not\models \bot$ (since $[\![\bot]\!]\alpha = 0$). Therefore, $\Gamma \not\models \bot$.

"$\Rightarrow$": Assume $\Gamma \not\models \bot$. This means it is not true that for all assignments $\alpha$ such that $\alpha \models \Gamma$ it follows that $\alpha \models \bot$. Therefore, there exists an assignment $\alpha$ such that $\alpha \models \Gamma$, but $\alpha \not\models \bot$. In particular, there exists an assignment such that $\alpha \models \Gamma$, which means that $\Gamma$ is satisfiable.

## 5.13 Satisfiability Theorem

A set of formulas is formally consistent if and only it is satisfiable.

**Proof.** Let $\Gamma$ be a set of formulas. Using Lemma 5.12 above we can rephrase the statement we have to prove equivalently as

$$\Gamma \nvdash_c \bot \text{ if and only if } \Gamma \not\models \bot$$

or, equivalently, negating both sides, as

$$\Gamma \vdash_{\mathrm{c}} \bot \text{ if and only if } \Gamma \models \bot.$$

But this is the Soundness and and Completeness Theorem (7.7.1, 7.7.2) for $A := \bot$.

*Remark.* If $\Gamma$ is satisfiable (that is $\Gamma \not\models \bot$) one often also says that $\Gamma$ is **semantically consistent**. Hence the Satisfiability Theorem can also be phrased as follows:

A set of formulas is formally consistent if and only if it is semantically consistent.

Since formal consistency and semantic consistency coincides one often omits the predicate "formal" and "semantic" and just speaks of "consistency".

## 5.14 Exercise

Prove the Compactness Theorem from the Soundness and Completeness Theorem.

**Solution.** Let $\Gamma$ be a set of formulas such that every finite subset of $\Gamma$ is satisfiable. By Lemma 5.12, this means that for every finite subset $\Gamma_0$ of $\Gamma$ we have $\Gamma_0 \not\models \bot$. By the Soundness Theorem 7.7.1, it follows that for every finite subset $\Gamma_0$ of $\Gamma$ we have $\Gamma_0 \nvdash_{\mathrm{c}} \bot$. Therefore $\Gamma \nvdash_{\mathrm{c}} \bot$, since if we had a derivation of $\bot$ from $\Gamma$, this derivation would be finite and would hence use only finitely many formulas in $\Gamma$ as assumptions which would mean that $\Gamma_0 \vdash_{\mathrm{c}} \bot$ for some finite subset $\Gamma_0$ of $\Gamma$. From the Completeness Theorem 7.7.2 it follows that $\Gamma \not\models \bot$ which, by Lemma 5.12, means that $\Gamma$ is satisfiable.

## 5.15 Exercises

Derive the following formulas

**1. Minimal logic**

(a) $(A \to \neg B) \to (B \to \neg A)$

(b) $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

(c) $A \to \neg\neg A$

(d) $\neg(A \land \neg A)$

(e) $(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$

(f) $(A \vee B) \to \neg(\neg A \wedge \neg B)$

(g) $\neg(A \leftrightarrow \neg A)$

(f) $A \to \neg(A \wedge B) \to (C \to B) \to \neg C$ (essentially the "John the teacher" example)

**2. Intuitionistic logic**

(a) $(A \wedge \neg A) \to C$

(b) $(\neg A \vee B) \to (A \to B)$

(c) $(\neg\neg A \to A) \leftrightarrow ((\neg A \to A) \to A)$

(d) $(A \vee B) \to (\neg A \to B)$

**3. Classical logic**

(a) $\neg\neg A \to A$

(b) $(\neg A \to A) \to A$

(c) $A \vee \neg A$

(d) $\neg(\neg A \wedge \neg B) \to (A \vee B)$

(e) $\neg(\neg A \vee \neg B) \to (A \wedge B)$

(f) $\neg(A \to B) \to A \wedge \neg B$

**Sample Solution:** We give a derivation for 3 (f):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{u : (A \to B) \to \bot \qquad \dfrac{\dfrac{\dfrac{v : A \to \bot \qquad w : A}{\bot}\,{\to^-}}{B}\,\text{efq}}{A \to B}\,{\to^+}\; w : A}{\bot}\,{\to^-}}{(A \to \bot) \to \bot}\,{\to^+}\; v : A \to \bot}{A}\,\text{raa} \qquad \dfrac{\dfrac{u : (A \to B) \to \bot \qquad \dfrac{x : B}{A \to B}\,{\to^+}\; y : A}{\bot}\,{\to^-}}{B \to \bot}\,{\to^+}\; x : B}{A \wedge (B \to \bot)}\,{\wedge^+}}{((A \to B) \to \bot) \to (A \wedge (B \to \bot))}\,{\to^+}\; u : (A \to B) \to \bot$$

# 6 Resolution

This chapter follows closely the text book [6].

Resolution is a proof method used in logic programming and automatic theorem proving. A resolution proof shows that a given formula in CNF is unsatisfiable. Resolution is defined for formulas in CNF only, and it has only one proof rule.

Recall that a literal is an atomic proposition, or the negation of an atomic proposition. For any literal $L$ we define the *dual (or opposite or complementary) literal* $\overline{L}$ as

$$\overline{L} := \begin{cases} \neg A & \text{if } L = A \text{ (an atomic proposition)} \\ A & \text{if } L = \neg A \end{cases}$$

By a *clause* we mean in the following always a *disjunctive clause*, that is, a disjunction of literals. It is convenient to view a clause simply as a (finite) *set* of literals (connected implicitly by $\vee$).

We consider a formula in conjunctive normal form (CNF) as a (finite) set of clauses (connected implicitly by $\wedge$). Hence, in the following "set of clauses" and "CNF formula" means the same.

## 6.1 Example

An example of a CNF formula written in set notation is

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

where $A, B, C, D, E$ are atomic formulas. This means

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

The set notation implies that one ignores the order of literals and clauses as well as any duplications of literals or clauses. For example, the CNF formula

$$\{\{A, B\}, \{A, \neg B, A\}, \{\neg B, \neg B, A\}\}$$

is the same as

$$\{\{\neg B, A\}, \{B, A\}\}$$

This means that

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

is identified with

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

## 6.2 Definition (Resolvent)

Let $C_1$ and $C_2$ be (disjunctive) clauses. A clause $C$ is called a *resolvent* of $C_1$ and $C_2$ if there is a literal $L \in C_1$ such that $\overline{L} \in C_2$ and

$$C = (C_1 \setminus \{L\}) \cup (C_2 \setminus \{\overline{L}\})$$

We also write

$$\frac{C_1 \qquad C_2}{C}\,\mathrm{R}$$

to indicate that $C$ is a resolvent of $C_1$ and $C_2$. For example

$$\frac{A \vee B \vee \neg C \qquad \neg B \vee \neg C}{A \vee \neg C}\,\mathrm{R}$$

Since $\neg A \vee B$ is equivalent to $A \to B$, the following special cases of resolution

$$\frac{\neg A \vee B \qquad \neg B \vee C}{\neg A \vee C}\,\mathrm{R} \qquad\qquad \frac{\neg A \vee B \qquad A}{B}\,\mathrm{R}$$

correspond to transitivity of implication and modus ponens (or implication elimination), respectively

$$\frac{A \to B \qquad B \to C}{A \to C} \qquad\qquad \frac{A \to B \qquad A}{B} \to^-$$

## 6.3 Exercise

Compute all resolvents that can be generated from clauses in the CNF formula $F$ of Example 6.1.

A clause it is called *trivial (or tautological)* if it contains opposite literals. For example the clauses $A \vee E \vee \neg B$ and $\neg A \vee B \vee C$ have the resolvent $B \vee C \vee E \vee \neg B$ which is a trivial clause since it contains the opposite literals $B$ and $\neg B$. Trivial clauses are tautologies, that is, equivalent to $\top$, and can therefore deleted from a CNF without changing its meaning (because $\top \wedge A \equiv A$ for every formula $A$).

A clause $C_1$ is said to *subsume* a clause $C_2$ if $C_1$ is a subset of $C_2$. For example $A \vee \neg C$ subsumes $A \vee B \vee \neg C \vee D$. Clauses that are subsumed by another clause can be deleted without changing the meaning of the CNF (because $C \wedge (C \vee A)$ is equivalent to $C$). We identify a CNF with the equivalent CNF where all clauses that are trivial or subsumed by another clause are deleted. For example the CNF $A \wedge (B \vee C) \wedge (A \vee \neg B)$ is identified with $A \wedge (B \vee C)$

A clause is called a *unit clause* if it contains exactly one literal. If we resolve two unit clauses, say $\{A\}$ and $\{\neg A\}$, we obtain the empty clause as resolvent which corresponds to an empty disjunction. Since the empty disjunction is, by convention, defined as the false proposition, we denote it by $\perp$.

$$\frac{A \qquad \neg A}{\perp}\,\mathrm{R}$$

Since the empty clause is unsatisfiable, any CNF formula containing the empty clause is unsatisfiable.

Given a CNF $F$ (that is, a set of clauses) one tries to show that $F$ is unsatisfiable by deriving the empty clause from it through repeated resolution steps.

## 6.4 Resolution Lemma

Let $F$ be a CNF formula. Let $C$ be a resolvent of two clauses in $C_1$ and $C_2$ in $F$. Then $F$ and $F \cup \{C\}$ are equivalent.

**Proof.** One can either prove this by showing directly that $F$ and $F \cup \{C\}$ have the same value for all assignments, or give a formal proof that $F$ implies $C$. For the latter it suffices to show that, for arbitrary formulas $C, D, A$ we have that $C \vee A$ and $D \vee \neg A$ together imply $C \vee D$. This holds even in intuitionistic logic:

$$\vdash_{\mathrm{i}} (C \vee A) \to (D \vee \neg A) \to (C \vee D)$$

We leave this as an exercise.

## 6.5 Definition (Resolution Derivation)

A *resolution derivation* or (*proof*) of the empty clause from a clause set $F$ is a sequence $C_1, C_2, \ldots, C_m$ of different clauses such that

> $C_m$ is the empty clause, and for every $i = 1, \ldots, m$, $C_i$ either is a clause in $F$ or a resolvent of two clauses $C_a, C_b$ with $a, b \leq i$.

## 6.6 Theorem (Soundness for Resolution Derivations)

If there is a resolution derivation of the empty clause from a clause set $F$, then $F$ is unsatisfiable.

**Proof.**

This is a direct consequence of the Resolution Lemma (6.4). If the empty clause is derived from $F$, then the empty clause follows from $F$. Therefore, any satisfying assignment for $F$ would satisfy the empty clause, which is impossible.

## 6.7 Theorem (Completeness for Resolution Derivations)

If a clause set $F$ is unsatisfiable, then there is a resolution derivation of the empty clause from $F$.

**Proof (Sketch).**

If the empty clause is in $F$, then the empty clause is a trivial resolution derivation of of the empty clause from $F$.

Therefore, we assume in the following that $F$ does not contain the empty clause.

Let the *excess* of $F$ be the number of occurrences of literals in $F$ minus the number of clauses in $F$.

The proof proceeds by induction on the excess of $F$.

If the excess of $F$ is 0 then all clauses in $F$ are unit clauses. Since $F$ is unsatisfiable, there must be two clauses with opposite literals. Resolving these two clause yields the empty clause, and we are done.

Now assume the excess of $F$ is greater than 0. Since $F$ does not contain the empty clause, there must be at least one nonunit clause, say $C$. We select a literal $L$ form $C$ and form new clauses $C_0 := C \setminus \{L\}$. and $C_1 := \{L\}$.

Since $F$ is unsatisfiable and $C_0$ subsumes $C$, the clause set $F_0 := (F \setminus \{C\}) \cup \{C_0\}$ is unsatisfiable as well. Moreover, the excess of $F_0$ is smaller than the excess of $F$. Therefore, by induction hypothesis, there is a resolution derivation $d_0$ of the empty clause from $F_0$.

Similarly, the clause set $F_1 := (F \setminus \{C\}) \cup \{C_1\}$ is unsatisfiable, and, since its excess is smaller than the excess of $F$, there is, by induction hypothesis, a resolution derivation of the empty clause from $F_0$.

If $C_0$ is not used in $d_0$, then $d_0$ is in fact a derivation of the empty clause from $F$ and we are done.

Otherwise, we add $L$ back into $C_0$ and all its descendants in $d_0$ and get a new derivation $d_0'$, which is a derivation from $F$. If $d_0'$ still ends with the empty clause, then we are done. Otherwise, $d_0'$ must end with the clause $\{L\}(= C_1)$.

Now we can append $d_1$ at the end of $d_0'$ and obtain a derivation from $F$ of the empty clause.

## 6.8 Example

Let $F = (A \vee B \vee \neg C) \wedge \neg A \wedge (A \vee B \vee C) \wedge (A \vee \neg B)$. We prove that $F$ is unsatisfiable by the following resolution derivation (a comment $(i, j)$ means "resolvent of clauses $i$ and $j$", no comment means "clause from $F$"):

| | | |
|---|---|---|
| 1 | $A \vee B \vee \neg C$ | |
| 2 | $A \vee B \vee C$ | |
| 3 | $A \vee B$ | (1,2) |
| 4 | $A \vee \neg B$ | |
| 5 | $A$ | (3,4) |
| 6 | $\neg A$ | |
| 7 | $\perp$ | (5,6) |

*Remark.* In some cases resolution derivations can be found rather quickly, but there do exist examples for unsatisfiable formulas where exponentially many resolvents have to be generated before the empty clause is derived. (cf. A. Urquhart. Hard examples for resolution. *Journal of the Association of Computing Machinery* 34(1987):209-219).

## 6.9 Exercise (2-SAT)

Show that resolution for CNFs whose clauses have length at most two terminates in cubic time.

**Solution.** A resolvent of clauses of lengths $\leq 2$ has again length $\leq 2$ and contains only atoms that were contained in the resolved clauses. Since the number of clauses of length $\leq 2$ that can be formed from $n$ atoms is $(2n)^2$, this means that starting with a CNF whose clauses have length $\leq 2$ resolution will fail to produce new clauses (and hence terminate) after a quadratic number of resolution steps. Since the cost of one resolution step (finding the resolvents and performing the resolution) is bounded by the size of the given CNF, this results in a cubic run time.

## 6.10 Exercise (Linear resolution for Horn Clauses)

Show that the following modification of the resolution calculus, called *linear resolution*, is sound and complete for the class of Horn formulas:

> Derive a resolvent from two clauses only if one of them is a unit clause and drop the other clause.

Conclude that satisfiability for Horn formulas is decidable in quadratic time.

**Solution.** Let $H$ be a Horn formula. If $H$ is unsatisfiable, then it must contain either the empty clause, or else a positive unit clause that can be resolved with another clause. Otherwise, the assignment that assigns 1 to exactly those atoms that appear as (positive) unit clauses in $H$ satisfies $H$. Resolving a unit clause in $H$ with another clause $C$ in $H$ results in a resolvent that subsumes $C$. Hence $C$ can be safely deleted. This means that each such resolution step reduces the size of the Horn formula. Therefore linear resolution must terminate after a linear number of resolution steps resulting in a quadratic run time. After termination, the resulting Horn formula is, by the previous considerations, unsatisfiable if and only if it contains the empty clause.

## 6.11 DPLL

The Davis Putnam Logemann Loveland (DPLL) proof system is another method to show the unsatifiability of CNFs. It derives sequents of the form $\Gamma \vdash F$ where $\Gamma$ is a set of literals (the literals that are assumed to be true) and $F$ is a CNF, viewed as a set of clauses (and a clause is viewed as a set of literals). The meaning of $\Gamma \vdash F$ is that $\Gamma$ and $F$ are incompatible, that is, if all literals in $\Gamma$ are true, that $F$ is false. The special case that $\Gamma$ is empty means that $F$ is unsatisfiable.

The DPLL proof system consists of five rules (recall that $\overline{L}$ is the opposite of the literal $L$):

$$\frac{\Gamma, L \vdash F}{\Gamma \vdash F, \{L\}}\,\text{(Unit)} \qquad \frac{\Gamma, L \vdash F, C}{\Gamma, L \vdash F, (C, \overline{L})}\,\text{(Red)} \qquad \frac{\Gamma, L \vdash F}{\Gamma, L \vdash F, (C, L)}\,\text{(Elim)}$$

$$\frac{}{\Gamma \vdash F, \emptyset}\,\text{(Conflict)} \qquad \frac{\Gamma, L \vdash F \qquad \Gamma, \overline{L} \vdash F}{\Gamma \vdash F}\,\text{(Split)}$$

The rules are usually applied in a backwards fashion, that is, one moves from the conclusion of a rule to its premises.

The most interesting rules are (Unit), (Split) and (Conflict).

(Unit) is the *Unit Propagation Rule* which says that in order to satisfy a CNF the single literal in a unit clause must be true.

(Split) is the *Splitting Rule* that allows to select a literal $l$ and consider both cases when $l$ is true or false. This the most costly rule since it introduces a branching of the proof search resulting in a doubling of the search space. Therefore, this rule is applied only if no other rule can be applied

(Conflict) means that the CNF contains an empty clauses and is therefore unsatisfiable. Hence there is nothing further to do.

The rules (Red) and (Elim) are for are for simplification and elimination of clauses. It can be easily shown that DPLL is sound and complete and equivalent to Resolution.

## 6.12 SAT solvers and their applications

Resolution and DPLL and variants thereof are the dominant proof systems implemented in modern SAT solvers, that is, programs that decide whether a given CNF is unsatisfiable.

Today SAT solving can be found almost everywhere in computing, for example in electronic design automation, formal verification of software and hardware, planning and scheduling.

## 6.13 Example (Pythagorean triples)

A spectacular application of SAT solving in mathematics has recently been obtained by Oliver Kullmann (Swansea University), Marijn Heule (University of Texas at Austin) and Victor Marek (University of Kentucky). They solved the long-standing open *Pythagorean Triples Problem* ("Solving and Verifying the Boolean Pythagorean Triples problem via Cube-and-Conquer". arXiv:1605.00723):

A Pythagorean triple is a triple of positive integers $a, b, c$ such that $a^2 + b^2 = c^2$, for example $3, 4, 5$ (since $3^2 + 4^2 = 5^2$).

The Pythagorean Triples Problem asks whether it is possible to colour each of the positive integers $1, 2, ...$ either blue or red such that no Pythagorean triple $a, b, c$, are all the same color.

This problem was open for 30 years and was solved by Kullman, Heule and Marek negatively by showing the impossibility of such a colouring.

In fact, they showed that for the numbers $1, 2, \dots, 7825$ no colouring exists that avoids uniformly coloured Pythagorean triples in that range, while for the numbers $1, 2, \dots, 7824$ such a colouring does exist.

The proof was found using a SAT solver which generated a proof that is is 200 terabytes large and said to be the largest proof ever (see `http://www.nature.com/news/two-hundred-terabyte-maths-proof-is-largest-ever-1.19990`).

The Pythagorean triples problem (for a given number, say 7825) can be coded into a CNF as follows. For each number $a \in \{1, \dots, 7825\}$ one has a variable $B_a$ meaning that $a$ is coloured blue (then $\neg B_a$ means that $a$ is coloured red).

For each Pythagorean triple $a, b, c$ one writes down two clauses expressing that the numbers $a, b, c$ are not all the same color, that is, (at least) one of them must be blue and one must be red:

$$(B_a \vee B_b \vee B_c) \wedge (\neg B_a \vee \neg B_b \vee \neg B_c)$$

The CNF consisting of the all these clauses expresses that no Pythagorean triple is uni-coloured. Its unsatisfiability means that such a colouring is impossible.

This formula is a 3-CNF with 7825 variables and $2 * 9472 = 18944$ clauses (since there are 9472 Pythagorean triples below 7825).

Note that there are $2^{7825} \approx 10^{2365}$ assignments for this formula. Therefore, the naive method of checking all assignments is completely beyond reach (the number of atoms in the universe is in the area of $10^{80}$).

## 6.14 Exercise

Write down the CNF representing the Pythagorean triples problem for $\{1, \dots, 10\}$.

## 6.15 Exercise

Transform the negation of Peirce's formula, $\neg(((A \to B) \to A) \to A)$ into an equivalent CNF and show its unsatisfiability by a resolution derivation of the empty clause.

# 7 Predicate logic

In propositional logic we can only talk about the truth and falsity of *propositions*. If we want to talk about *objects* and their relationships, or *quantify* objects by saying "for every object" or "there exists an object", we need a more expressive logic called *predicate logic* also known as *first-order predicate logic* or *first-order logic*.

For example, the statement

"every positive number has a positive square root"

can be written in more detail (but still informally)

"for every number $x$, if $0 < x$, then there exists $y$ such that $0 < y$ and $x = y * y$"

As a formula in predicate logic this reads

$$\forall x\,(0 < x \to \exists y\,(0 < y \land x = y * y))$$

The subformulas $0 < x$, $0 < y$, and $x = y * y$ are called *atomic formulas*. The infix notation is syntactic sugar for $<(0, x)$, $<(0, y)$, and $x = *(y, y)$. $<$ is a *predicate symbol* (denoting the less-than relation), 0 is a *constant*, and $*$ is a *function symbol* (denoting multiplication). $x$ and $y$ are *variables*.

## 7.1 Examples and exercises

Before studying the precise syntax, semantics and proof calculus for predicate logic we discuss, informally, some simple examples in order to familiarise ourselves with the new concepts.

### 7.1.1 Exercise

Formalise the following statements in predicate logic

| | |
|---|---|
| $u$ : | If the train is late and there is no taxi at the station, then I'm late for the meeting |
| $v$ : | The train is late |
| $w$ : | I'm not late for the meeting |

using the atomic formulas

| | |
|---|---|
| $TL$ | The train is late. |
| $TS(x)$ | Taxi $x$ is at the station |
| $LM$ | I'm late for the meeting |

Prove the statement “There is a taxi at the station” from the assumptions $u, v, w$.

Discuss whether the use of predicate logic was necessary in this example.

**Solution.**

| | |
|---|---|
| $u$ : | $TL \wedge \neg\exists x\, TS(x) \to LM$ |
| $v$ : | $TL$ |
| $w$ : | $\neg LM$ |

Informal proof: We have to show $\exists x TS(x)$. We do a proof by contradiction. We assume

$p$ : $\neg\exists x TS(x)$,

aiming for a contradiction. By the assumptions $u$, $v$ and $p$ we get $LM$. But this contradicts the assumption $w$.

We can formalise this proof using only the proof rules of propositional logic.

$$
\begin{prooftree}
\AxiomC{$w : \neg LM$}
\AxiomC{$u : TL \wedge \neg\exists x\, TS(x) \to LM$}
\AxiomC{$v : TL$}
\AxiomC{$p : \neg\exists x\, TS(x)$}
\RightLabel{$\wedge^+$}
\BinaryInfC{$TL \wedge \neg\exists x\, TS(x)$}
\RightLabel{$\to^-$}
\BinaryInfC{$LM$}
\RightLabel{$\to^-$}
\BinaryInfC{$\bot$}
\RightLabel{$\to^+, p : \neg\exists x\, TS(x)$}
\UnaryInfC{$\neg\neg\exists x\, TS(x)$}
\RightLabel{raa}
\UnaryInfC{$\exists x\, TS(x)$}
\end{prooftree}
$$

This means that the use of predicate logic was not necessary: We could have encapsulated the predicate logic formula $\exists x\, TS(x)$ by treating it as an atomic proposition ignoring its internal structure since it is not relevant for the proof.

### 7.1.2 Exercise

Formalise the statements

| | |
|---|---|
| $u$ : | All people living in Snowdonia speak Welsh |
| $v$ : | John doesn’t speak Welsh |

using the atomic formulas

| | |
|---|---|
| $S(x)$ | Person $x$ lives in Snowdonia. |
| $W(x)$ | Person $x$ speaks Welsh. |

Prove the statement “John doesn’t live in Snowdonia” from the assumptions $u, v$.

Try to isolate a new proof rule that was necessary to complete the proof.

**Solution.**

$$\begin{array}{ll} u: & \forall x\,(S(x) \to W(x)) \\ v: & \neg W(John) \end{array}$$

We have to prove $\neg S(John)$ from the assumptions $u$ and $v$. To this end we assume

$w: \quad S(John)$,

aiming for a contradiction. Specializing the assumption $u$ to $x = John$, we obtain

$$S(John) \to W(John).$$

Using the assumption $w : S(John)$ we conclude $W(John)$. But this contradicts the assumption $v : \neg W(John)$.

In order to give a fully formal proof we introduce a new proof rule, called "for all elimination" ($\forall^-$), that allows to specialize a given universally quantified statement.

$$\dfrac{\forall x\, A(x)}{A(t)}\,\forall^-$$

Here $t$ is a *term*, that is, a formal representation of an object. In our case $t$ is the constant *John*.

$$\dfrac{\dfrac{v:\neg W(John) \qquad \dfrac{\dfrac{u:\forall x\,(S(x)\to W(x))}{S(John)\to W(John)}\,\forall^- \qquad w:S(John)}{v:W(John)}\,\to^-}{\bot}\,\to^-}{\neg S(John)}\,\to^+, w:S(John)$$

### 7.1.3 Exercise

Formalise the statement

Every female who has the same father as me, is a sister of mine.

in two different ways:

(a) using a relation for father-ship,

(b) using a function symbol for denoting the father of a person.

**Solution.**

(a) $\forall x\, (Female(x) \land \exists y\, (Father(y, x) \land Father(y, me)) \to Sister(x, me)$

(b) $\forall x\, (Female(x) \land father(x) = father(me) \to Sister(x, me)$

### 7.1.4 Exercise

Formalise the following statements:

(a) An American astronaut is well trained.

(b) An American astronaut has landed on the moon.

(c) Nobody danced.

The solution is left as an exercise in the lecture.

## 7.2 Syntax of predicate logic

As illustrated by the previous examples, predicate logic deals not only with propositions, but also with objects and relations between objects. The syntactic entities to denote objects are called *terms* the syntactic entities to denote relations are called *predicates*. Terms are built from *variables* and *function symbols*. Each function symbol has an *arity* which specifies what sort of arguments it accepts and what sort of results it produces. Similarly, each predicate symbol has an *arity* which specifying what sort of arguments it accepts.

### 7.2.1 Definition (Signature)

A **many-sorted signature** (**signature** for short), is a triple $\Sigma = (S, \mathcal{F}, \mathcal{P})$ such that the following conditions are satisfied.

- $S$ is a nonempty set. The elements $s \in S$ are called **sorts**.
- $\mathcal{F}$ is a set whose elements are called **function symbols**, and which are of the form
  $$f\colon s_1 \times \ldots \times s_n \to s,$$
  where $n \geq 0$ and $s_1, \ldots, s_n, s \in S$.

  $s_1 \times \ldots \times s_n \to s$ is called **arity** of $f$, with **argument sorts** $s_1, \ldots, s_n$ and **target sort** $s$.

  Function symbols are also called *operations* or *functions*.

  A function symbol of the form $c\colon \to s$ (i.e. $n = 0$) is called a **constant** of sort $s$. For constants we often use the shorter notation $c\colon s$ (i.e. we omit the arrow).
- $\mathcal{P}$ is a set whose elements are called **predicate symbols**, and which are of the form
  $$P\colon (s_1 \times \ldots \times s_n),$$
  where $n \geq 0$ and $s_1, \ldots, s_n \in S$.

  $(s_1 \times \ldots \times s_n)$ is called **arity** of $P$, with **argument sorts** $s_1, \ldots, s_n$.

  Predicate symbols are also called *predicates*.

  Note that atomic propositions can be viewed as predicate symbols of zero arguments. In this sense, propositional logic can be viewed as a part of predicate logic.

A sort $s$ is to regarded as a name of a set, so it is similar to a *type* in programming. A function symbol $f\colon s_1 \times \ldots \times s_n \to s$ is a name for a function on the sets denoted by the sorts $s_1, \ldots, s_n, s$. This will be made more precise when we discuss the *semantics* of predicate logic.

In logicians jargon a signature is also called a *many-sorted first-order language* and by the "arity" of a function or predicate symbol one also means just the number of arguments it takes. If the arity is 1, 2, or 3, one speaks of unary, binary, or ternary function or predicate symbols.

### 7.2.2 Example

Consider the signature $\Sigma := (S, \mathcal{F}, \mathcal{P})$, where

$$\begin{array}{rcl} S & = & \{\mathsf{nat}\} \\ \mathcal{F} & = & \{0\colon \mathsf{nat}, +\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}\} \\ \mathcal{P} & = & \{<\colon (\mathsf{nat} \times \mathsf{nat}), \leq\colon (\mathsf{nat} \times \mathsf{nat})\} \end{array}$$

The function symbols of a signature $\Sigma$ can be used to build formal expressions, called *terms*, which denote elements of a given structure interpreting the signature $\Sigma$ (the notion of a structure will be made precise later).

### 7.2.3 Definition (Term)

Let $\Sigma = (S, \mathcal{F}, \mathcal{P})$ be a signature, and let $X = (X_s)_{s \in S}$ be a family of pairwise disjoint sets. The elements of $X_s$ are called **variables** of sort $s$. We define **terms** and their sorts by the following rules.

(i) Every variable $x \in X_s$ is a term of sort $s$.

(ii) Every constant $c$ in $\Sigma$ of sort $s$ is a term of sort $s$.

(iii) If $f\colon s_1 \times \ldots \times s_n \to s$ is a function symbol in $\Sigma$, and $t_1, \ldots, t_n$ are (previously defined) terms of sorts $s_1, \ldots, s_n$, respectively, then the formal expression

$$f(t_1, \ldots, t_n)$$

is a term of sort $s$.

The set of all terms of sort $s$ is denoted by $\mathrm{T}(\Sigma, X)_s$.

A term is **closed** if it doesn't contain variables, i.e. is built without the use of rule (i).

The set of all closed terms of sort $s$ is denoted by $\mathrm{T}(\Sigma)_s$. Clearly $\mathrm{T}(\Sigma)_s = \mathrm{T}(\Sigma, \emptyset)_s$.

### 7.2.4 Example

For the signature $\Sigma$ of example 7.4.3 and the set of variables $X := \{x, y\}$ the following are examples of terms in $\mathrm{T}(\Sigma, X)$:

$x$

$0$

$0 + y$ (which stands for $+(0, y)$)

$(0 + x) + y$ (which stands for $+(+(0, x), y)$)

$((0 + 0) + (x + x)$ (which stands for $+(+(0, 0), +(x, x))$

$0 + (0 + (0 + 0))$ (which stands for $+(0, +(0, +(0, 0)))$

The second and the last of these terms are closed.

In a similar way as terms are syntactic constructs denoting objects, formulas are syntactic construct to denote propositions.

### 7.2.5 Definition (Formula)

The set of **formulas** over a signature $\Sigma = (S, \mathcal{F}, \mathcal{P})$ and a set of variables $X = (X_s)_{s \in S}$ is defined inductively by the following rules.

(i) $\top$ and $\bot$ are formulas.

(ii) $t_1 = t_2$ is a formula, called **equation**, for each pair of terms $t_1, t_2 \in \mathrm{T}(\Sigma, X)$ of the same sort.

(iii) If $P\colon (s_1 \times \ldots \times s_n)$ is a predicate symbol in $\mathcal{P}$ and $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, the $P(t_1, \ldots, t_n)$ is a formula.

(iv) If $A$ and $B$ are formulas then $(A \wedge B)$, $(A \vee B)$ and $(A \to B)$ are formulas.

(v) If $A$ is a formula then $(\forall x\, A)$ and $(\exists x\, A)$ are formulas for every variable $x \in X$, called **universal quantification** ('for all') and **existential quantification** ('exists'), respectively.

Formulas over a signature $\Sigma$ are also called **$\Sigma$-formulas**

A **free occurrence** of a variable $x$ in a formula $A$ is an occurrence of $x$ in $A$ which is not in the scope of a quantifier $\forall x$ or $\exists x$. We let $\mathsf{FV}(A)$ denote the set of free variables of $A$, i.e. the set of variables with a free occurrence in $A$. A formula $A$ is **closed** if $\mathsf{FV}(A) = \emptyset$.

We set

$$\mathcal{L}(\Sigma, X) := \{A \mid A \text{ is a } \Sigma\text{-formula }, \mathsf{FV}(A) \subseteq X\}$$

and use the abbreviation

$$\mathcal{L}(\Sigma) := \mathcal{L}(\Sigma, \emptyset),$$

i.e. $\mathcal{L}(\Sigma)$ is the set of closed $\Sigma$-formulas.

### 7.2.6 Remarks and Notations

1. Formulas as defined above are usually called **first-order formulas**, since we allow quantification over object variables only. If we would also quantify over set variables we would obtain second-order formulas.

2. A formula is **quantifier free**, **qf** for short, if it doesn't contain quantifiers.

3. A formula is **universal** if it is of the form $\forall x_1 \ldots \forall x_n\, A$ where $A$ is quantifier free.

4. The *basic formulas* are $\bot$, equations $t_1 = t_2$ and formulas of the form $P(t_1, \ldots, t_n)$. for the latter we often write more briefly $P(\vec{t})$.

### 7.2.7 Abbreviations

As in the case of propositional logic we omit brackets as long as this does not lead to ambiguities. We let the quantifiers bind stronger than the logical connectives, hence $\forall x\, A \to B$ stands for $(\forall x\, A) \to B$. Furthermore, we use the following abbreviations:

| Formula | Abbreviation |
|---|---|
| $A \to \bot$ | $\neg A$ (negation) |
| $\forall x_1 \forall x_2 \ldots \forall x_n\, A$ | $\forall x_1, x_2, \ldots, x_n A$ |
| $\exists x_1 \exists x_2 \ldots \exists x_n\, A$ | $\exists x_1, x_2, \ldots, x_n\, A$ |
| $\forall x_1, \ldots, x_n A$, where $\{x_1, \ldots, x_n\} = \mathsf{FV}(A)$ | $\forall\, A$ (closure of $A$) |
| $(A \to B) \wedge (B \to A)$ | $A \leftrightarrow B$ (equivalence) |

### 7.2.8 Examples

$$A \quad :\equiv \quad x = 0 \to y + x = y$$

$$B \quad :\equiv \quad \exists y\, (x = y \to \forall z\, x = z)$$

$$C \quad :\equiv \quad \forall x\, (0 \leq x)$$

$A$ is quantifier free. $C$ is universal. $A$ and $C$ are universal. $\mathsf{FV}(A) = \{x, y\}$, $\mathsf{FV}(B) = \{x\}$.

### 7.2.9 Definition (Substitution)

By $A[x := t]$ we denote the result of substituting in the formula $A$ every free occurrence of the variable $x$ by the term $t$, possibly renaming bound variables in $A$ in order to avoid variable clashes.

For example,

$$\begin{aligned} & (\forall x\,(0 < x \to y < x + y) \land \forall y\,(\neg y < 0))[y := x + 1] \\ = \quad & \forall z\,(0 < z \to x + 1 < z + (x + 1)) \land \forall y\,(\neg y < 0) \end{aligned}$$

Note that we renamed the bound variable $x$ by the fresh variable $z$. For $z$ to be "fresh" means that $z$ occurs neither free in the formula nor does it occur in the term $t := x + 1$. Note also that the occurrence of $y$ in the subformula $\neg y < 0$ is not substituted because it is bound by $\forall y$ .

## 7.3 Formalising statements in predicate logic

We now look at further examples of informal statements and their formalisations in predicate logic. Since we haven't yet introduced the precise semantics of predicate logic (this will be done in Sect. 7.4) we have to rely on an intuitive understanding of the meaning of formulas.

### 7.3.1 Example (Students)

Formalise the following statements in predicate logic:

(a) All students submitted Coursework 1 or Coursework 2.

(b) All students who submitted Coursework 2 also submitted Coursework 1.

(c) Not all students attended all lectures.

(d) No student skipped all lectures.

(e) No two students attended exactly the same lectures.

Use the following signature, i.e. variables, constants function symbols and predicates, in your formalisation:

| | |
|---|---|
| $x, y$ | students |
| $l$ | lectures |
| $\mathrm{CW1}(x)$ | student $x$ submitted Coursework 1 |
| $\mathrm{CW2}(x)$ | student $x$ submitted Coursework 2 |
| $\mathrm{A}(x, l)$ | student $x$ attended lecture $l$ |

**Solution.**

(a) $\forall x\,(\mathrm{CW1}(x) \vee \mathrm{CW2}(x))$

(b) $\forall x\,(\mathrm{CW2}(x) \to \mathrm{CW1}(x))$

The remaining questions are left as an exercise for the lecture.

(c)

(d)

(e)

### 7.3.2 Example (Maths)

Formalise the following statements in predicate logic:

(a) The equation $x^2 + 1 = 0$ has no solution.

(b) Exactly the non-negative numbers have a square root.

(c) $\sqrt{2}$ is irrational.

(d) A square root of an integer is either an integer or irrational.

Use the following signature:

| | |
|---|---|
| $x, y, z$ | real numbers |
| $0, 1$ | constants 0 and 1 |
| $+, *$ | addition and multiplications (used infix) |
| $\mathrm{Q}(x)$ | $x$ is a rational number |
| $\mathrm{Z}(x)$ | $x$ is an integer |
| $x \leq y$ | $x$ is less or equal $y$ (used infix) |

You may use abbreviation, for example $2 := 1 + 1$, $x^2 := x * x$, etc.

**Solution.**

(a) $\neg\exists x(x^2 + 1 = 0)$

(b) $\forall x\,(\exists y(y^2 = x) \leftrightarrow 0 \leq x)$

The remaining questions are left as an exercise for the lecture.

(c)

(d)

### 7.3.3 Example (Processes)

Consider a system of processes and two binary relations $R$ and $S$ on on processes.
If $R(x, y)$ holds, we say $x$ *reduces* $y$, or $y$ is a *reduct* of $x$.
If $S(x, y)$ holds, we say $y$ *simulates* $x$.
The relation $R$ is called

- *deterministic* if every process can be reduced in at most one way;
- *confluent* if whenever $x$ reduces to $y$ and $z$, then $y$ and $z$ have a common reduct;
- *deadlock free* if every process can be reduced;
- *normalising* if every process reduces to a process that cannot be reduced further.

The relation $S$ is called a *bisimulation for $R$* if whenever $y$ simulates $x$, then all reducts of $x$ are simulated by some reduct of $y$ and all reducts of $y$ simulate some reduct of $x$.

(a) Formalise each of the properties of $R$ described informally above.

(b) Formalise the property that $S$ is a bisimulation for $R$.

**Solution.**

(a) - *deterministic*: $\forall x\,\forall y\,\forall z\,(R(x, y) \land R(x, z) \to y = z)$.

The remaining questions are left as an exercise for the lecture.

- *confluent*:
- *deadlock free*:
- *normalising*:

(b)

## 7.4 Semantics of predicate logic

In order to determine the truth value of a formula $F$ in *propositional logic* it was sufficient to have an assignment $\alpha$ prescribing to every atomic proposition $A$ a truth value $\alpha(A) \in \{0,1\}$. For *predicate logic* we need in addition a *universe*, that is a set of objects, and assignments must prescribe meanings to

| | |
|---|---|
| constants | as elements of the universe |
| function symbols | as functions on the universe |
| predicate symbols | as boolean valued functions on the universe |

### 7.4.1 Definition (Structure)

A **structure** $\mathcal{M} = ((\mathcal{M}_s)_{s \in S}, \alpha)$ for a signature $\Sigma = (S, \mathcal{F}, \mathcal{P})$ is given by the following:

- For each sort $s$ in $S$ a *nonempty* set $\mathcal{M}_s$, called the **carrier set** of sort $s$.
- An assignment $\alpha$ that assigns
  - to each constant $c\colon s$ in $\mathcal{F}$ an element $\alpha(c) \in \mathcal{M}_s$,
  - to each function symbol $f\colon s_1 \times \ldots \times s_n \to s$ in $\mathcal{F}$ a *function*

$$\alpha(f)\colon \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n} \to \mathcal{M}_s$$

  - For each predicate symbol $P\colon (s_1 \times \ldots \times s_n)$ in $\mathcal{P}$ a boolean valued function (also called *predicate* or *relation*)

$$\alpha(P)\colon \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n} \to \{0,1\}$$

### 7.4.2 Remarks

1. In the definition of a signature (7.2.1) the expression $f\colon s_1 \times \ldots \times s_n \to s$ is meant *symbolically*, i.e. '$\times$' and '$\to$' are to be read as uninterpreted symbols. In the definition of a structure (7.4.1), however, we used the familiar mathematical notation for set-theoretic functions to communicate by $\alpha(f)\colon \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n} \to \mathcal{M}_s$ a *semantical* object, namely a function $\alpha(f)$ whose domain is the cartesian product of the sets $\mathcal{M}_{s_i}$ and whose range is $\mathcal{M}_s$.

2. It is common to call the elements $\alpha(c)$ *constants*. Hence the word 'constant' has a double meaning. Conversely, the function symbols of a signature are often just called 'functions'. Whenever we use such slightly ambiguous names it will be clear from the context what is meant.

3. We will often identify the semantics of a predicate symbol, $\alpha(P)\colon \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n} \to \{0,1\}$, with the set $\{\vec{a} \mid \vec{a} \in \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n}, \alpha(P)(\vec{a}) = 1\} \subseteq \mathcal{M}_{s_1} \times \ldots \times \mathcal{M}_{s_n}$.

4. By a $\Sigma$-*structure* we mean a structure for the signature $\Sigma$.

5. Signatures with more than one sort are sometimes called *many-sorted signatures* and structures without predicate symbols are sometimes called *algebras*. This is the case for example in the book

[10] J V Tucker, Theory of Programming Languages, Course Notes, UWS, 2005.

### 7.4.3 Example

Consider the signature $\Sigma := (S, \mathcal{F}, \mathcal{P})$, where

$$\begin{array}{rcl} S &=& \{\mathsf{nat}\} \\ \mathcal{F} &=& \{0\colon \mathsf{nat}, +\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\ \mathcal{P} &=& \leq\colon (\mathsf{nat} \times \mathsf{nat})\} \end{array}$$

We follow [10] and display signatures in a box:

| | |
|---|---|
| **Signature** | $\Sigma$ |
| **Sorts** | $\mathsf{nat}$ |
| **Constants** | $0\colon \mathsf{nat}$ |
| **Functions** | $+\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$ |
| **Predicates** | $\leq\colon (\mathsf{nat} \times \mathsf{nat})$ |

The $\Sigma$-structure $\mathcal{M}$ of natural numbers with 0, and addition and the predicate $\leq$ is given by:

the carrier set $\mathbf{N} = \{0, 1, 2, \ldots\}$, that is,

$\mathcal{M}_{\mathsf{nat}} = \mathbf{N}$;

the constants 0, that is,

$\alpha(0) = 0$;

the operation of addition on $\mathbf{N}$, that is,

$\alpha(+)(n, m) = n + m$;

the relation $< \subseteq \mathbf{N} \times \mathbf{N}$, that is

$\alpha(\leq) = \{(n, m) \in \mathbf{N} \times \mathbf{N} \mid n \leq m\}$ (that is, $\alpha(\leq)(n, m) = 1$ iff $n \leq m$).

Again we use the more readable box notation [10]

| | |
|---|---|
| **Structure** | $\mathcal{M}$ |
| **Carriers** | $\mathbf{N}$ |
| **Constants** | $0$ |
| **Functions** | $+ \colon \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ |
| **Predicates** | $\leq \subseteq \mathbf{N} \times \mathbf{N}$ |

For the signature $\Sigma$ we may also consider another structure, $\mathcal{M}' = (\mathcal{M}'_{\mathsf{nat}}, \alpha')$ with carrier $\mathcal{M}'_{\mathsf{nat}} = \mathbf{N}^+ := \mathbf{N} \setminus \{0\}$ $(= \{1, 2, 3, 4, \ldots\})$, the constant 1, multiplication restricted to $\mathbf{N}^+$, and the divisibility relation $\cdot \mid \cdot$. Hence we have

the carrier set $\mathbf{N}^+ = \{1, 2, \ldots\}$, that is,

$\mathcal{M}'_{\mathsf{nat}} = \mathbf{N}^+$;

the constants 1, that is,

$\alpha'(0) = 1$;

the operation of multiplication on $\mathbf{N}^+$, that is,

$\alpha'(*)(n, m) = n * m$;

the relation $\cdot \mid \cdot \subseteq \mathbf{N} \times \mathbf{N}$, that is

$$\alpha'(\leq) = \{(n,m) \in \mathbf{N}^+ \times \mathbf{N}^+ \mid \ n \text{ divides } m\}.$$

Written in box notation:

| | |
|---|---|
| **Structure** | $\mathcal{M}'$ |
| **Carriers** | $\mathbf{N}^+$ |
| **Constants** | $1$ |
| **Functions** | $*\colon \mathbf{N}^+ \times \mathbf{N}^+ \to \mathbf{N}^+$ |
| **Predicates** | $\cdot \vert \cdot \subseteq \mathbf{N}^+ \times \mathbf{N}^+$ |

### 7.4.4 Remarks

1. The display of signatures and structures via boxes has to be handled with some care. If, for example, in the box displaying the signature $\Sigma$ in example 7.4.3 we would have exchanged the order of the sorts nat and boole, we would have defined the same signature. But then the box displaying the structure $\mathcal{M}$ would not be well-defined, since then the sort boole would be associated with the set $\mathbf{N}$ and nat with $\mathbf{B}$, and consequently the arities of the operations of $\Sigma$ would not fit with the operations of the structure $\mathcal{M}$.

Therefore: When displaying signatures and structures in boxes *order matters*.

### 7.4.5 Definition (Semantics of terms)

Let $\mathcal{M} = ((\mathcal{M}_s)_{s\in S}, \alpha)$ be a structure for the signature $\Sigma = (S, \mathcal{F}, \mathcal{P})$, and let $X = (X_s)_{s\in S}$ a set of variables.

A **variable assignment** $\eta\colon X \to \mathcal{M}$ is a function assigning to every variable $x \in X_s$ an element $\eta(x) \in \mathcal{M}_s$.

Given a variable assignment $\eta\colon X \to \mathcal{M}$ we define for each term $t \in \mathrm{T}(\Sigma, X)_s$ its **value**

$$t^{\mathcal{M},\eta} \in \mathcal{M}_s$$

by the following rules.

(i) $x^{\mathcal{M},\eta} := \eta(x)$.

(ii) $c^{\mathcal{M},\eta} := \alpha(c)$.

(iii) $f(t_1, \ldots, t_n)^{\mathcal{M},\eta} := \alpha(f)(t_1^{\mathcal{M},\eta}, \ldots, t_n^{\mathcal{M},\eta})$.

For closed terms $t$, i.e. $t \in \mathrm{T}(\Sigma)$ $(= \mathrm{T}(\Sigma, \emptyset))$ the variable assignment $\eta$ and rule (i) are obsolete and we write $t^{\mathcal{M}}$ instead of $t^{\mathcal{M},\eta}$.

### 7.4.6 Exercise

Let $\Sigma$ be the signature of example 7.4.3 and $\mathcal{M}$ the $\Sigma$-structure of the natural numbers with zero, addition and the 'less-or-equal' relation. Write down $\Sigma$-formulas expressing in $\mathcal{M}$ the following statements.

(a) $x$ is an even number.

(b) $x$ is greater than $y$.

(c) $x$ is the average of $y$ and $z$.

In order to precisely declare the **semantics** of a formula we define what it means for a formula to be true in a structure.

### 7.4.7 Definition (Semantics of logical connectives and quantifiers)

In order to define the semantics of formulas we need the semantics of the logical connectives and quantifiers. The semantics of the logical connectives $(\wedge, \vee, \rightarrow)$ was defined in Section 2.5 by the boolean functions

$$\wedge, \vee, \rightarrow : \{0,1\}^2 \rightarrow \{0,1\}$$

It remains to define

$$\exists, \forall : \mathcal{P}_+(\{0,1\}) \rightarrow \{0,1\}$$

where $\mathcal{P}_+(\{0,1\})$ is the set of all non-empty subsets of {0,1}, i.e. $\mathcal{P}_+(\{0,1\}) = \{\{0\},\{0,1\},\{1\}\}$. We define for every $X \in \mathcal{P}_+(\{0,1\})$

$$\exists(X) := \begin{cases} 1 & \text{if } 1 \in X \\ 0 & \text{otherwise} \end{cases}$$

$$\forall(X) := \begin{cases} 1 & \text{if } 0 \notin X \\ 0 & \text{otherwise} \end{cases}$$

Hence,

$$\begin{array}{ll} \exists(X) = 1 & \text{if and only if there exists } x \in X \text{ such that } x = 1 \\ \forall(X) = 1 & \text{if and only if for all } x \in X,\ x = 1 \end{array}$$

As truth tables this reads

| $X$ | $\exists(X)$ |
|---|---|
| $\{0\}$ | 0 |
| $\{0,1\}$ | 1 |
| $\{1\}$ | 1 |

| $X$ | $\forall(X)$ |
|---|---|
| $\{0\}$ | 0 |
| $\{0,1\}$ | 0 |
| $\{1\}$ | 1 |

### 7.4.8 Definition (Semantics of formulas)

Let $\Sigma = (S, \mathcal{F}, \mathcal{P})$ be a signature, $X = (X_s)_{s \in S}$ a set of variables, $\mathcal{M} = ((\mathcal{M}_s)_{s \in S}, \alpha)$ a $\Sigma$-structure, $\eta \colon X \to \mathcal{M}$ a variable assignment, and $A \in \mathcal{L}(\Sigma, X)$ a $\Sigma$-formula.

For every formula $A$ of predicate logic we define the value

$$[\![A]\!](\mathcal{M}, \eta) \in \{0, 1\}$$

$$\begin{array}{rcl}
[\![\top]\!](\mathcal{M}, \eta) & = & 1 \\
[\![\bot]\!](\mathcal{M}, \eta) & = & 0 \\
[\![t_1 = t_2]\!](\mathcal{M}, \eta) & = & \begin{cases} 1 & \text{if } t_1^{\mathcal{M},\eta} = t_2^{\mathcal{M},\eta} \\ 0 & \text{otherwise} \end{cases} \\
[\![P(t_1, \ldots, t_n)]\!](\mathcal{M}, \eta) & = & \alpha(P)(t_1^{\mathcal{M},\eta}, \ldots, t_1^{\mathcal{M},\eta}) \\
[\![A \diamond B]\!](\mathcal{M}, \eta) & = & \diamond([\![A]\!](\mathcal{M}, \eta), [\![B]\!](\mathcal{M}, \eta)) \quad \text{for } \diamond \in \{\wedge, \vee, \to\} \\
[\![Q\, x\, A]\!](\mathcal{M}, \eta) & = & Q(\{[\![A]\!](\mathcal{M}, \eta[x := a]) \mid a \in \mathcal{M}_s\}) \quad \text{for } Q \in \{\forall, \exists\}
\end{array}$$

where in the last case the variable $x$ is assumed to be of sort $s$. We also define

$$\mathcal{M}, \eta \models A :\Leftrightarrow [\![A]\!](\mathcal{M}, \eta) = 1$$

which is to be read '$A$ **is true in** $\mathcal{M}$ **under** $\eta$', or '$\mathcal{M}, \eta$ **is a model of** $A$'.

The following Lemma is an immediate consequence of the definition above. In many textbooks the clauses of the Lemma are used to *define* the semantics of formulas.

### 7.4.9 Lemma (Semantics of formulas)

(i) $\mathcal{M}, \eta \models \top$.

(ii) $\mathcal{M}, \eta \not\models \bot$, i.e. $\mathcal{M}, \eta \models \bot$ does not hold.

(iii) $\mathcal{M}, \eta \models t_1 = t_2$ iff $t_1^{\mathcal{M},\eta} = t_2^{\mathcal{M},\eta}$.

(iv) $\mathcal{M}, \eta \models P(t_1, \ldots, t_n)$ iff $\alpha(P)(t_1^{\mathcal{M},\eta}, \ldots, t_n^{\mathcal{M},\eta}) = 1$.

(v) $\mathcal{M}, \eta \models A \wedge B$ iff $\mathcal{M}, \eta \models A$ and $\mathcal{M}, \eta \models B$.

$\mathcal{M}, \eta \models A \vee B$ iff $\mathcal{M}, \eta \models A$ or $\mathcal{M}, \eta \models B$.

$\mathcal{M}, \eta \models A \to B$ iff $\mathcal{M}, \eta \models A$ implies $\mathcal{M}, \eta \models B$ (i.e. $\mathcal{M}, \eta \not\models A$ or $\mathcal{M}, \eta \models B$).

(vi) $\mathcal{M}, \eta \models \forall x\, A$ iff $\mathcal{M}, \eta[x := a] \models A$ for all $a \in \mathcal{M}_s$ (provided $x$ is of sort $s$).

$\mathcal{M}, \eta \models \exists x\, A$ iff $\mathcal{M}, \eta[x := a] \models A$ for at least one $a \in \mathcal{M}_s$ (provided $x$ is of sort $s$).

For closed $\Sigma$-formulas $A$ the variable assignment is obviously redundant and we write

$$\mathcal{M} \models A$$

for $\mathcal{M}, \eta \models A$. For a set $\Gamma$ of closed $\Sigma$-formulas we say that the $\Sigma$-structure $\mathcal{M}$ is a **model of** $\Gamma$, written

$$\mathcal{M} \models \Gamma,$$

if $\mathcal{M} \models A$ for all $A \in \Gamma$.

We can now extend the notions 'logical consequence', 'logical validity', and 'satisfiability', which we know from propositional logic, to predicate logic.

### 7.4.10 Definition (Logical consequence)

Let $\Gamma$ be a set of closed formulas and $A$ a closed formula. We say that $A$ is a **logical consequence** of $\Gamma$, or $\Gamma$ **logically implies** $\mathcal{M}$, written

$$\Gamma \models A,$$

if $A$ is true in all models of $\Gamma$, that is,

$$\mathcal{M} \models \Gamma \quad \text{implies} \quad \mathcal{M} \models A, \qquad \text{for all } \Sigma\text{-structures } \mathcal{M}$$

### 7.4.11 Definition (Logical validity)

A closed $\Sigma$-formula $A$ is said to be **(logically) valid**, written

$$\models A,$$

if $A$ is true in all $\Sigma$-structures, that is $\mathcal{M} \models A$ for all $\Sigma$-structures $\mathcal{M}$. Valid formulas is also called a **tautologies**.

Obviously, $A$ is valid if and only if it is a logical consequence of the empty set of formulas.

### 7.4.12 Definition (Satisfiability)

A set of closed $\Sigma$-formulas $\Gamma$ is called **satisfiable** if it has a model, that is, there exists a $\Sigma$-structure $\mathcal{M}$ in which all formulas of $\Gamma$ are true ($\mathcal{M} \models \Gamma$).

### 7.4.13 Exercise

Show that validity and satisfiability are related by the following equivalences:

$$\begin{array}{rcl} A \text{ valid} & \Leftrightarrow & \{\neg A\} \text{ unsatisfiable (that is, not satisfiable)} \\ A \text{ satisfiable} & \Leftrightarrow & \{\neg A\} \text{ not valid} \end{array}$$

### 7.4.14 Definition (equivalence)

We call two $\Sigma$-formulas $A, B$ *equivalent*, written $A \equiv B$, if they always have the same truth value, that is, for every $\Sigma$-structure $\mathcal{M}$ and every variable assignment $\eta : X \to \mathcal{M}$

$$[\![A]\!](\mathcal{M}, \eta) = [\![B]\!](\mathcal{M}, \eta)$$

This is the same as saying that for every $\Sigma$-structure $\mathcal{M}$

$$\mathcal{M} \models A \quad \text{if and only if} \quad \mathcal{M} \models B$$

and also to saying that the formula $A \leftrightarrow B$ is logically valid.

Clearly, '$A \equiv B$' is an equivalence relation, that is, the following laws hold.

*Reflexivity*: $A \equiv A$ for every formula $A$.

*Symmetry*: If $A \equiv B$ then $B \equiv A$.

*Transitivity*: If $A \equiv B$ and $B \equiv C$ then $A \equiv C$.

Here is a list of useful equivalences (recall that $\neg A$ stands for $A \to \bot$):

$$\begin{array}{rcl} \exists x\, A(x) & \equiv & \neg \forall x\, \neg A(x) \\ \forall x\, A(x) & \equiv & \neg \exists x\, \neg A(x) \\ \neg \exists x\, A(x) & \equiv & \forall x\, \neg A(x) \\ \neg \forall x\, A(x) & \equiv & \exists x\, \neg A(x) \end{array}$$

The first two equivalences state that each of the quantifiers can be expressed by the other. The third and the fourth equivalence state that negation can be "pushed inside".

### 7.4.15 Exercise (Negation normal form)

The statement

Just because nobody complains doesn't mean all parachutes are perfect

can be formalized as

$$\neg(\neg\,\exists x\, complain(x) \to \forall p\,(parachute(p) \to perfect(p)))$$

Transform this formula into an equivalent one where negation is only applied to atomic formulas (such formulas are called in *negation normal form*).

**Solution.**

$$\begin{array}{ll} & \neg(\neg\,\exists x\, complain(x) \to \forall p\,(parachute(p) \to perfect(p))) \\ \equiv & \neg\exists x\, complain(x) \land \neg\,\forall p\,(parachute(p) \to perfect(p)) \\ \equiv & \forall x\,\neg complain(x) \land \exists p\,\neg(parachute(p) \to perfect(p)) \\ \equiv & \forall x\,\neg complain(x) \land \exists p\,(parachute(p) \land \neg\, perfect(p)) \end{array}$$

## 7.5 The Theorems of Church and Loewenheim-Skolem

We discuss two fundamental results regarding the decidability of truth and the size of models in predicate logic.

### 7.5.1 Theorem (A Church, 1935 and A Turing, 1936)

It is undecidable whether or not a closed formula in predicate logic is valid.

This theorem can be proven by reducing the halting problem to the validity problem (i.e. coding Turing machines into logic).

Another proof is by showing that Post's correspondence problem, which is known to be undecidable, can be encoded by predicate logic.

Although, by the Church-Turing Theorem, the validity problem is undecidable, there is an effective procedure generating all valid formulas (technically: the set of valid formulas is recursively enumerable). We will study such a generation process in the next chapter.

Recall that for propositional logic the situation is different: logical validity for formulas in propositional logic is decidable. More precisely, the set of logically valid propositional formulas is **co-NP** complete, since its complement is the set of propositional formulas $A$ for which $\neg A$ is satisfiable, which is **NP** complete.

Another fundamental result which concerns satisfiability is the

### 7.5.2 Loewenheim-Skolem Theorem

Every countable satisfiable set of closed formulas has a countable model.

A surprising consequence of the Loewenheim-Skolem Theorem is the fact there is a countable structure that satisfies the same predicate logic formulas as the (uncountable!) structure of real numbers.

### 7.5.3 Examples

Consider a signature with the sorts $\mathsf{nat}$ and $\mathsf{boole}$ and the operation $<: \mathsf{nat} \times \mathsf{nat} \to \mathsf{boole}$. Then the formula

$$\exists x\,\forall y\,(x < y) \to \forall y\,\exists x\,(x < y)$$

is a tautology. The formula

$$\forall x, y\,(x < y \to \exists z\,(x < z \land z < y))$$

is satisfiable, but not a tautology (why?). Set

$$\Gamma := \{\forall x\,\neg(x < x),\ \forall x.y.z\,(x < y \land y < z \to x < z)\}$$

Then the formula

$$A :\equiv \forall x, y\,(x < y \to \neg y < x)$$

is a logical consequence of $\Gamma$, that is, $\Gamma \models A$.

| | Introduction rules | Elimination rules |
|---|---|---|
| $\forall$ | $\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x\, A(x)}\ \forall^+$ (*) | $\dfrac{\Gamma \vdash \forall x\, A(x)}{\Gamma \vdash A(t)}\ \forall^-$ |
| $\exists$ | $\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x\, A(x)}\ \exists^+$ | $\dfrac{\Gamma \vdash \exists x\, A(x) \qquad \Gamma \vdash \forall x\,(A(x) \to B)}{\Gamma \vdash B}\ \exists^-$ (**) |

Figure 3: The rules of natural deduction for quantifiers (sequent notation)

## 7.6 Natural deduction proofs for predicate logic

In addition to the rules for propositional logic, the Natural Deduction calculus has introduction and elimination rules for the quantifiers:

(*) The $\forall^+$ rule is subject to the so-called *variable condition*: $x$ must not occur free in $\Gamma$.

(**) The $\exists^-$ rule is subject to the restriction that $x$ must not be free in $B$.

Figure 4 shows the rules for quantifiers in the short notation where the assumptions $\Gamma$ are implicit.

| | Introduction rules | Elimination rules |
|---|---|---|
| $\forall$ | $\dfrac{A(x)}{\forall x\, A(x)}\ \forall^+$ (*) | $\dfrac{\forall x\, A(x)}{A(t)}\ \forall^-$ |
| $\exists$ | $\dfrac{A(t)}{\exists x\, A(x)}\ \exists^+$ | $\dfrac{\exists x\, A(x) \qquad \forall x\,(A(x) \to B)}{B}\ \exists^-$ (**) |

Figure 4: The rules of natural deduction for quantifiers (short notation)

With the short notation the variable condition for $\forall^+$ reads:

(*) $x$ must not occur free in in any free assumption valid at that point.

### 7.6.1 Examples

1. In the following derivation of $\forall y\, A(y+1)$ from $\forall x\, A(x)$ we use the *for-all introduction rule*, $\forall^+$, and the *for-all elimination rule*, $\forall^-$:

$$\dfrac{\dfrac{\forall x\, A(x)}{A(x+1)}\,\forall^-}{\forall x\, A(x+1)}\,\forall^+$$

In the application of $\forall^+$ the variable condition is satisfied, because $x$ is not free in $\forall x\, A(x)$.

2. Find out what's wrong with the following 'derivations'.

$$\dfrac{\dfrac{\forall y(x < 1+y)}{x < 1+0}\,\forall^-}{\forall x(x < 1+0)}\,\forall^+$$

$$\dfrac{\dfrac{\dfrac{\forall\, x(\forall y(x < y+1) \to x = 0)}{\forall y(y < y+1) \to y = 0}\,\forall^- \qquad \forall y(y < y+1)}{y = 0}\,\to^-}{\forall y(y = 0)}\,\forall^+$$

3. The *exists introduction rule*, $\exists^+$, and the *exists elimination rule*, $\exists^-$ are used in the following derivation.

$$
\begin{prooftree}
\AxiomC{$u : \forall x\,(x-1)+1 = x$}
\RightLabel{$\forall^-$}
\UnaryInfC{$(x-1)+1 = x$}
\RightLabel{$\exists^+$}
\UnaryInfC{$\exists y(y+1 = x)$}
\RightLabel{$\forall^+$}
\UnaryInfC{$\forall x\,\exists y(y+1 = x)$}
\RightLabel{$\to^+ u$}
\UnaryInfC{$\forall x\,((x-1)+1 = x) \to \forall x\,\exists y(y+1 = x)$}
\end{prooftree}
$$

4. Let us derive from the assumptions $\exists x\,A(x)$ and $\forall x(A(x) \to B(f(x)))$ the formula $\exists y\,B(y)$:

$$
\begin{prooftree}
\AxiomC{$\exists x\,A(x)$}
\AxiomC{$\forall x(A(x) \to B(f(x)))$}
\RightLabel{$\forall^-$}
\UnaryInfC{$A(x) \to B(f(x))$}
\AxiomC{$u : A(x)$}
\RightLabel{$\to^-$}
\BinaryInfC{$A(f(x))$}
\RightLabel{$\exists^+$}
\UnaryInfC{$\exists y\,B(y)$}
\RightLabel{$\to^+ u$}
\UnaryInfC{$A(x) \to \exists y\,B(y)$}
\RightLabel{$\forall^+$}
\UnaryInfC{$\forall x\,(A(x) \to \exists y\,B(y))$}
\RightLabel{$\exists^-$}
\BinaryInfC{$\exists y\,B(y)$}
\end{prooftree}
$$

We see that in the application of $\forall^+$ the variable condition is satisfied, because $x$ is not free in $\exists y\,B(y)$.

### 7.6.2 Equality rules

So far we only considered the Natural Deduction rules for logic without equality. Here are the rules for equality:

**Reflexivity** $$\dfrac{}{t = t}\ \text{refl}$$

**Symmetry** $$\dfrac{s = t}{t = s}\ \text{sym}$$

**Transitivity** $$\dfrac{r = s \qquad s = t}{r = t}\ \text{trans}$$

**Compatibility** $$\dfrac{s_1 = t_1 \qquad \ldots \qquad s_n = t_n}{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}\ \text{comp}$$

for every operation $f$ of $n$ arguments

$$\dfrac{s_1 = t_1 \qquad \ldots \qquad s_n = t_n \qquad P(s_1, \ldots, s_n)}{P(t_1, \ldots, t_n)}\ \text{comp}$$

for every predicate $P$ of $n$ arguments

### 7.6.3 Example

Let us derive from the assumptions $\forall x, y\,(x + y = y + x)$ and $\forall x\,(x + 0 = x)$ the formula $\forall x, y\,((0 + x) * y = x * y)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x\,\forall y\,(x + y = y + x)}{\forall y\,(0 + y = y + 0)}\ \forall^-}{0 + x = x + 0}\ \forall^- \qquad \dfrac{\forall x\,(x + 0 = x)}{x + 0 = x}\ \forall^-}{0 + x = x}\ \text{trans} \qquad \dfrac{}{y = y}\ \text{refl}}{(0 + x) * y = x * y}\ \text{comp}}{\forall y\,((0 + x) * y = x * y)}\ \forall^+}{\forall x\,\forall y\,((0 + x) * y = x * y)}\ \forall^+$$

### 7.6.4 Definition (Classical, intuitionistic, minimal logic)

The notions of minimal, intuitionistic and classical derivability for predicate logic completely analogous to the propositional case (Definition 5.6), but for formulas in predicate logic and for derivations composed from propositional rules and the new rules for quantifiers and equality:

Let $\Gamma$ be a (possibly infinite) set of formulas and $A$ a formula, all in predicate logic.

$\Gamma \vdash_{\mathrm{c}} A \quad :\Leftrightarrow \quad \Gamma_0 \vdash A$ is derivable for some finite subset $\Gamma_0$ of $\Gamma$.

($A$ is derivable from $\Gamma$ in *classical logic*)

$\Gamma \vdash_{\mathrm{i}} A \quad :\Leftrightarrow \quad \Gamma \vdash_{\mathrm{c}} A$ with a derivation not using the rule reductio-ad-absurdum.

($A$ is derivable from $\Gamma$ in *intuitionistic logic*)

$\Gamma \vdash_{\mathrm{m}} A \quad :\Leftrightarrow \quad \Gamma \vdash_{\mathrm{c}} A$ with a derivation using neither the rule reductio-ad-absurdum nor the rule ex-falso-quodlibet.

($A$ is derivable from $\Gamma$ in *minimal logic*)

### 7.6.5 Lemma

Let $t(x)$ be a term possibly containing the variable $x$, and let $r, s$ be terms of the same sort as $x$. Then

$$r = s \vdash_{\mathrm{m}} t(r) = t(s)$$

**Proof.** Induction on $t(x)$.

If $t(x)$ is a constant or a variable different from $x$, then $t(r)$ and $t(r)$ are the same term $t$. Hence the assertion is $r = s \vdash_{\mathrm{m}} t = t$ which is an instance of the reflexivity rule.

If $t(x)$ is the variable $x$ then $t(r)$ is $r$ and $t(s)$ is $s$, and the assertion becomes $r = s \vdash_{\mathrm{m}} r = s$ which is an instance of the assumption rule.

Finally, consider $t(x)$ of the form $f(t_1(x), \ldots, t_n(x))$. By induction hypothesis we may assume that we already have a derivation of $r = s \vdash_{\mathrm{m}} t_i(r) = t_i(s)$ for $i = 1, \ldots, n$. One application of the compatibility rule yields the required sequent.

### 7.6.6 Lemma

Let $A(x)$ be a formula possibly containing the variable $x$, and let $r, s$ be terms of the same sort as $x$. Then:

$$r = s \vdash_{\mathrm{m}} A(r) \leftrightarrow A(s)$$

**Proof.** Induction on the formula $A(x)$.

If $A(x)$ is an equation, say, $t_1(x) = t_2(x)$, then we have to derive

$$r = s \vdash_{\mathrm{m}} t_1(r) = t_2(r) \leftrightarrow t_1(s) = t_2(s)$$

By Lemma 7.6.5 we have already derivations of

$$r = s \vdash_{\mathrm{m}} t_1(r) = t_1(s) \qquad \text{and} \qquad r = s \vdash_{\mathrm{m}} t_2(r) = t_2(s)$$

It is now easy to obtain the required derivation using the symmetry rule and the transitivity rule. We leave this as an exercise to the reader.

If $A(x)$ is a compound formula we can use the induction hypothesis in a straightforward way.

## 7.7 Soundness and completeness

The soundness and completeness theorems for predicate logic are analogous to the propositional case. However, now the notion of a model is much more complex. While in the propositional case a model was just a vector of Booleans (assigning truth values to atomic propositions), in the case of predicate logic a model is a first-order structure which is a rather complex object.

### 7.7.1 Soundness Theorem for predicate logic

If $\Gamma \vdash_{\mathrm{c}} A$ then $\Gamma \models A$.

**Proof.** The proof is easy and similar to the propositional case, but taking into account the additional rules for quantifiers .

### 7.7.2 Completeness Theorem for predicate logic (K Gödel, 1929)

If $\Gamma \models A$ then $\Gamma \vdash_{\mathrm{c}} A$.

In words: If $A$ is a logical consequence of $\Gamma$ (i.e. $A$ is true in all models of $\Gamma$), then this can be formally derived by the inference rules of natural deduction for predicate logic.

The proof of this theorem is completely different from the corresponding proof for the propositional case and beyond the scope of this course. Detailed expositions can be found in any textbook on Mathematical Logic. The original proof is part of Gödel's doctoral dissertation.

The notion of formal consistency is analogous to the propositional case:.

### 7.7.3 Definition (Formal consistency for predicate logic)

A (possibly infinite) set of formulas $\Gamma$ is called **formally consistent** if $\Gamma \nvdash_c \bot$, that is there is no (classical) derivation of $\bot$ from assumptions in $\Gamma$.

In other words: A set of formulas $\Gamma$ is formally consistent if and only if no contradiction can be derived from $\Gamma$.

### 7.7.4 Satisfiability Theorem for predicate logic

Every consistent set of formulas has a model.

**Proof.** As for the propositional case.

Another important consequence of Gödel's Completeness Theorem is the fact that all logically valid formulas can be effectively enumerated.

### 7.7.5 Enumerability Theorem for predicate logic

The set of all logically valid formulas is recursively enumerable, that is, there is an effective procedure producing all logically valid formulas.

**Proof.** By the Soundness and Completeness Theorem, the logically valid formulas are exactly the provable ones. Hence, it suffices to systematically generate all proofs.

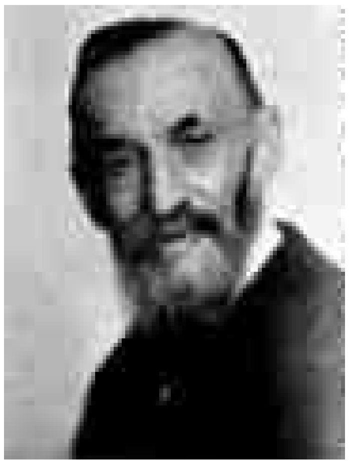
By the Church-Turing Theorem (Theorem 7.5.1) we know that logically validity in predicate logic is undecidable. Therefore, effective enumerability for logically valid formulas in predicate logic is the best one can achieve.

## 7.8 Axioms and rules for natural numbers and other data types

For many common data types we can formulate axioms describing their characteristic features. We will only treat the (unary) natural numbers. Similar axioms could be stated for binary number, lists, finite trees etc., more generally for freely generated data types.

### 7.8.1 Peano Axioms

The following axioms and rules where introduced (in a slightly different form) by Peano to describe the structure of natural numbers with zero and the successor function (we write $t+1$ for the successor of $t$).



G Peano (1858 - 1932)

In the following the terms $s, t$ and the variable $x$ are supposed to be of sort `nat`.

**Peano 1** $$\frac{}{0 \neq t+1}\,\text{peano1}$$

**Peano 2** $$\frac{}{s+1=t+1 \to s=t}\,\text{peano2}$$

**Induction** $$\frac{A(0) \qquad \forall x\,(A(x) \to A(x+1))}{\forall x\,A(x)}\,\text{ind}$$

### 7.8.2 Remark

In applications there will be further axioms describing additional operations on the natural numbers. Examples are, the equations defining addition and multiplication by *primitive recursion*:

$$\begin{aligned} x + 0 &= x \\ x + (y + 1) &= (x + y) + 1 \end{aligned}$$

$$\begin{aligned} x * y &= 0 \\ x * (y + 1) &= x * y + x \end{aligned}$$

At least as famous as the Completeness Theorem for Predicate Logic is Gödel's *(First) Incompleteness Theorem* for the theory of natural numbers given by the Peano Axioms (*On Formally Undecidable Propositions of Principia Mathematica and Related Systems, Monatshefte für Mathematik und Physik 38, 1931*).

### 7.8.3 Incompleteness Theorem (K Gödel, 1931)

Every consistent and effectively presented system of axioms and rules extending the Peano Axioms is incomplete, that is, there exists a formula $A$ such that neither $A$ nor its negation $\neg A$ is provable.

In particular, there exists a true statement about natural numbers that is not provable.

Similar Incompleteness Theorems hold for lists and trees and other data types that can (in principle) be encoded by natural numbers.

# 8 Outlook: Other logics

In this concluding chapter we very briefly describe some other logics that are very important in Computer Science. Many of these logics are extensions of propositional logic or predicate logic. A good background reading for this chapter is the book [10].

## 8.1 Second-Order Logic

Predicate logic is sometimes called *First-Order Logic (FOL)* because one can quantify only over *objects*, which are called 'first-order' ($\forall x$, $\exists x$).

*Second-Order Logic (SOL)* allows in addition for quantification over *predicates* (or *sets* or *properties*). In order to express this, Second-Order Logic has so-called *second-order variables* $X, Y, \ldots$ ranging over predicates (in addition to the usual first-order variables $x, y, \ldots$). Furthermore, one can build formulas of the form

$\forall X\, A(X)$, meaning that $A(X)$ holds for all predicates $X$.

$\exists X\, A(X)$, meaning that $A(X)$ holds for some predicate $X$.

Second-Order Logic (abbreviated SOL) can express many concepts that are important in Computer Science, for example:

**Finiteness** There exist only finitely many $x$ such that …(see the exercise below).

**Reachability** Node $y$ is reachable from a node $x$ (in a given graph or network)

**Order completeness** Every non-empty bounded set has a least-upper bound (in a given ordered structure, for example the real numbers)

**Being a natural number** $x$ is a natural number, where $x$ ranges of real numbers (see the example below).

None of these concepts can be expressed in First-Order Logic. For example, in FOL one can say "there exist no more than two", "there exist no more than three" etc., but it is not possible to say "there exist only finitely many".

**Exercise.** Use the Compactness Theorem (which holds for FOL) to show that it is impossible to express "there exist only finitely many" in FOL.

More precisely, let $\Sigma$ be a signature containing a unary predicate $P$. Show that there is no $\Sigma$-formula $A$ such that for all $\Sigma$-structures $\mathcal{M}$,

$A$ is true in $\mathcal{M}$ if and only if $P(a)$ is true in $\mathcal{M}$ for finitely many $a$ only.

**Solution.** Assume such a formula $A$ exists. For $n \in \mathrm{N}$ let $B_n$ be a formula expressing that $P(x)$ holds for at least $n$ elements $x$:

$$B_n := \exists x_1 \ldots \exists x_n ( \bigwedge_{1 \leq i,j \leq n, i \neq j} x_i \neq x_j \land P(x_1) \land \ldots \land P(x_n))$$

(for example $C_3 = \exists x_1 \exists x_2 \exists x_3 \, (x_1 \neq x_2 \land x_1 \neq x_3 \land x_2 \neq x_3 \land P(x_1) \land P(x_2) \land P(x_3)))$.

Let $\Gamma := \{A, B_1, B_2, B_3, \ldots\}$. Clearly, every finite subset $\Gamma_0$ of $\Gamma$ is satisfiable: Just take a $\Sigma$-structure where $P$ is interpreted by a set with $n$ elements where $n$ is the largest number such that $B_n \in \Gamma_0$.

By the Compactness Theorem, $\Gamma$ is satisfiable by some $\Sigma$-structure $\mathcal{M}$. Because $\mathcal{M}$ satisfies all $B_1, B_2, B_3, \ldots$, $P(a)$ must hold by infinitely many elements of $\mathcal{M}$. But $\mathcal{M}$ also satisfies $A$ (since $A \in \Gamma$). This contradicts the assumption that all models of $A$ interpret $P$ as a finite set.

**Example.** Let $x$ be a variable ranging over real numbers. The property $N(x)$ expressing that $x$ is a natural number can be defined in SOL as follows:

$$N(x) := \forall X \, (X(0) \land \forall y \, (X(y) \to X(y+1))) \to X(x))$$

Moreover, with this definition the proof principle of induction over natural numbers

$$A(0) \land \forall x \, (A(x) \to A(x+1)) \to \forall x \, (N(x) \to A(x))$$

can be logically proven (and must not be stated as an axiom).

On the other hand many of the good properties of First-Order Logic do not hold for Second-Order Logic. For example, the Compactness Theorem, the Completeness and the Loewenheim-Skolem Theorem all fail for Second-Order Logic.

## 8.2 Temporal Logic

Temporal logic has been introduced to reason about time dependent statements. It was first considered by the logician and philosopher Arthur Prior (1914-1969). Today, temporal logic is used in computer science to describe and verify state dependent system. The main idea is that a formally is not statically true or false, like in Propositional Logic or Predicate Logic, but its truth value may be dynamically change as the state changes in time. Important publications are *A. Pnueli. The Temporal Logic of*

*Programs. 18th annual IEEE-CS Symposium on Foundations of Computer Science, pages 46-77, 1977*, and *M. Ben-Ari, Z. Manna, A. Pnueli. The Temporal Logic of Branching Time. ACM POPL'81, Acta Informatica vol 20, pages 207-226, 1983.* A good overview of the subject is given in *E.A. Emerson. Temporal and modal logic. Handbook of Theoretical Computer Science, Chapter 16. MIT Press, 1995.*

There are many different variants of temporal Logic which mainly differ in the model of time. In *Linear Temporal Logic (LTL)* time is modelled as a sequence of states that extends indefinitely into the future. Linear temporal Logic is an extension of propositional logic. In addition to atomic propositions, $\bot, \top$, conjunction, disjunction, implication and negation there are

> $\mathbf{X}\,A$, meaning that $A$ holds in the ne**X**t state.
>
> $\mathbf{F}\,A$, meaning that $A$ holds in some **F**uture state.
>
> $\mathbf{G}\,A$, meaning that $A$ holds **G**lobally, that is, in all future states.
>
> $A\,\mathbf{U}\,B\ A$, meaning that $A$ holds **U**ntil $B$ holds, and $B$ will hold eventually.

The symbols $\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{W}$ are called temporal connectives.

A *model* of LTL is a sequence $\alpha = \alpha_1, \alpha_2, \dots$ where each $\alpha_i$ is an assignment in the sense of propositional logic. For an atomic proposition $A$, $\alpha_i(A) = 1$ means that $A$ holds in state $i$. Usually one considers as models all paths through a given directed graph whose nodes are labelled by propositional logic assignments. Such graphs describe the possible state transitions of a given computational system. The value of an LTL formula $A$ in a model $\alpha$ at state $i$, written $[\![A]\!]\alpha, i$ is defined as follows:

$$
\begin{array}{rcl}
[\![A]\!](\alpha,i) & = & \alpha_i(A) \quad \text{(for atomic propositions } A\text{)} \\
[\![\top]\!](\alpha,i) & = & 1 \\
[\![\bot]\!](\alpha,i) & = & 0 \\
[\![\neg A]\!](\alpha,i) & = & \neg([\![A]\!](\alpha,i) \\
[\![A \diamond G]\!](\alpha,i) & = & \diamond([\![A]\!](\alpha,i),[\![G]\!](\alpha,i)) \quad \text{ for } \diamond \in \{\wedge,\vee,\to\} \\
[\![\mathbf{X}\,A]\!](\alpha,i) & = & [\![A]\!](\alpha,i+1) \\
[\![\mathbf{F}\,A]\!](\alpha,i) & = & 1 \text{ if } [\![A]\!](\alpha,j)=1 \text{ for some state } j \geq i, \\
 & & 0 \text{ otherwise} \\
[\![\mathbf{G}\,A]\!](\alpha,i) & = & 1 \text{ if } [\![A]\!](\alpha,j)=1 \text{ for all states } j \geq i, \\
 & & 0 \text{ otherwise} \\
[\![A\,\mathbf{U}\,B]\!](\alpha,i) & = & 1 \text{ if there is a state } k \geq i \text{ s.t. } [\![A]\!](\alpha,j)=1 \text{ for } i \leq j < k \text{ and } [\![B]\!](\alpha,k)=1, \\
 & & 0 \text{ otherwise}
\end{array}
$$

Models are often depicted in the form

$$\begin{array}{ccccccccccccccccc} 1 & & 2 & & 3 & & 4 & & 5 & & 6 & & 7 & & 8 & & \ldots \\ (A) & \to & () & \to & (B) & \to & (A,B) & \to & (B) & \to & (A,B) & \to & (B) & \to & (A,B) & \to & \ldots \end{array}$$

meaning that $A$ holds exactly at the states $1, 4, 6, 8, \ldots$ and $B$ holds exactly at the states $3, 4, 5, 6, 7, 8 \ldots$.

**Exercise.** Which of the following formulas hold at state 1 in the model shown above.

(a) $A$

(b) $B$

(c) $\mathbf{X}\, A$

(d) $\mathbf{X}\,\mathbf{G}\,(A \to B)$

(e) $\mathbf{G}\,\mathbf{F}\, A$ (this expresses that $A$ will be true infinitely often)

(f) $\mathbf{F}\,\mathbf{G}\, A$ (this expresses that from some point on $A$ will be always true)

(g) $\mathbf{F}\,\mathbf{G}\, B$

(h) $\mathbf{X}\,((\neg A)\,\mathbf{U}\, B)$

The temporal operators **F** and **G** are also known as *modalities* and are sometimes written $\Diamond$ and $\Box$. Hence Temporal Logic is an example of Modal Logic (see the section on Modal Logic).

## 8.3 Modal Logic

Modal Logics are used to reason about distributed, concurrent and multi-agent computing systems, but also to reason about knowledge and belief. In modal logic one has, in addition to propositional formulas, the formulas

$\Box A$ ($A$ is necessarily true)

$\Diamond A$ ($A$ is possibly true)

The precise meaning of the modalities $\Diamond$ and $\Box$ depends on the application. Some (simplified) examples:

| | $\Box A$ | $\Diamond A$ |
|---|---|---|
| Temporal logic | $A$ holds always in the future | $A$ holds sometime in the future |
| Deontic logic | $A$ must be done | $A$ may be done |
| Epistemic logic | $A$ is known | $A$ is believed |
| Process logic | $A$ holds in all accessible states | $A$ holds in some accessible state |

The most common interpretation of modal logic is through *Kripke models*, introduced by the logician and philosopher Saul Kripke (born 1940).

A Kripke model is a triple $(W, R, \Vdash)$ where:

$W$ is a set of *possible worlds*.

$R$ is an *accessibility relation* between worlds.

$\Vdash$ is a relation between worlds and formulas such that

$w \Vdash \neg A$ if and only if $w \nVdash A$,

$w \Vdash A \wedge B$ if and only if $w \Vdash A$ and $w \Vdash A$,

$w \Vdash A \vee B$ if and only if $w \Vdash A$ or $w \Vdash A$

$w \Vdash \Box A$ if and only if $v \Vdash A$ for *all* $v \in W$ such that $R(w, v)$,

$w \Vdash \Diamond A$ if and only if $v \Vdash A$ for *some* $v \in W$ such that $R(w, v)$.

A modal formula $A$ holds in a Kripke model $(W, R, \Vdash)$ if $w \Vdash \Diamond A$ holds for all $w \in W$.

A modal formula $A$ is *valid* if it holds in all Kripke models.

Often one restricts the class of Kripke models to characterise variants of modal logic. For example, one may consider only those Kripke models where the accessibility relation is transitive or linear.

It is easy to see that the formulas

$$\Box A \leftrightarrow \neg\Diamond\neg A \quad \text{and} \quad \Diamond A \leftrightarrow \neg\Box\neg A$$

are valid. Hence the each modality can be defined in terms of the other. One also says that the modalities are *dual* to each other (in the same way as the quantifiers $\forall x$ and $\exists x$ are dual to each other).

Modal logic is particularly attractive for computer science since on the one hand it can express many important properties of processes, but on the other hand validity is *decidable* (in contrast to predicate logic where validity is undecidable). Therefore, modal logic is used in many tools for automated program verification.

A good introduction into the history of modal logic is *R. Goldblatt. Mathematical Modal Logic: A view of its evolution. Handbook of the History of Logic, Vol 7, 2006.*

## 8.4 Hoare Logic

Hoare logic is a proof calculus for verifying imperative programs introduced by Tony Hoare in 1969 (An axiomatic basis for computer programming, Communications of the ACM 12 (10): 576580). It derives statements of the form

$$\{P\}\,S\,\{Q\}$$

called *Hoare triples*, where $P$ and $Q$ are formulas, and $S$ is a command, that is, a fragment of an imperative program. The formula $P$ is called *precondition* and the formula $Q$ is called *postcondition*.

The intuitive meaning of a Hoare triple $\{P\}\,S\,\{Q\}$ is:

> If $P$ holds before the execution of $S$ and the execution of $S$ terminates, then after termination $Q$ will hold.

An example of a valid Hoare triple is

$$\{x < 3\}\ x := 2 * x + 1\ \{x < 6\}$$

an an invalid one is

$$\{x < 3\}\ x := 2 * x + 1\ \{x > 4\}$$

(assuming $x$ is of natural number type).

Here are the core rules of Hoare logic:

$$\frac{}{\{P\}\,\mathbf{skip}\,\{P\}}$$

$$\frac{}{\{P[E/x]\}\,x := E\,\{P\}}$$

$$\frac{\{P\}\,S\,\{Q\} \qquad \{Q\}\,T\,\{R\}}{\{P\}\,S;T\,\{R\}}$$

$$\frac{\{B \wedge P\}\,S\,\{Q\} \qquad \{\neg B \wedge P\}\,T\,\{Q\}}{\{P\}\,\mathbf{if}\,B\,\mathbf{then}\,S\,\mathbf{else}\,T\,\{R\}}$$

$$\frac{P_1 \to P_2 \qquad \{P_2\}\,S\,\{Q_2\} \qquad Q_2 \to Q_1}{\{P_1\}\,S\,\{Q_1\}}$$

$$\frac{\{B \wedge P\}\,S\,\{P\}}{\{P\}\,\mathbf{while}\,B\,\mathbf{do}\,S\,\{\neg B \wedge P\}}$$

This interpretation of Hoare triples corresponds to *partial correctness* because validity of a triple does not guarantee that the program $S$ terminates.

For *total correctness* one would require that the execution of $S$ actually terminates. In this case Hoare triples are usually written with square brackets, $[P]\,S\,[Q]$, and the interpretation is

> If $P$ holds before the execution of $S$, then the execution of $S$ will terminate and after termination $Q$ will hold.

All rules above except for the while-rule are valid for total correctness. The while rule has to be modified as follows:

$$\frac{[B \land P \land t = z]\ S\ [P \land t < z]}{[P]\ \mathbf{while}\, B\, \mathbf{do}\, S\ [\neg B \land P]}$$

Here $t$ is an expression and $z$ a variable of natural number type. The premise of the rule guarantees that as long as $B$ holds, each execution of $S$ will decrease the value of $t$. Since this decrease can only happen a finite number of times, the execution of the while loop must terminate. The crucial mathematical property of the natural numbers used here is the fact that the ordering $<$ on the natural numbers is *wellfounded.* This means that there is no infinite decreasing chain $n_1 > n_2 > \ldots$ such that all $n_i$ are natural numbers. The total while-rule above is valid more generally for any wellfounded ordering.

## 8.5 Fuzzy Logic

*Fuzzy Logic* an example of a *Many Valued Logic*. The main idea is that instead of only the Boolean truth values 0 ("false") and 1 ("true"), any real number between 0 and 1 is a possible truth value of a formula. $[\![A]\!] = p$ can be read as "$A$ is true with probability $p$" or as "$A$ is true to a degree $p$". For example

| | | |
|---|---|---|
| $[\![\text{Rain}]\!] = 1$ | means | "it will certainly rain" |
| $[\![\text{Rain}]\!] = 0.1$ | means | "there is a 10% chance of rain" |

Another interpretation of $[\![A]\!] = p$ would be "$A$ is true to a degree $p$". For example

| | | |
|---|---|---|
| $[\![\text{Hot}]\!] = 1$ | means | "it is very hot" |
| $[\![\text{Hot}]\!] = 0.2$ | means | "it is rather cold" |

In fuzzy logic the truth tables for the logical connectives are replaced by mathematical formulas, such as

$$\begin{array}{rcl} \neg x & = & 1 - x \\ x \wedge y & = & \min(x, y) \\ x \vee y & = & \max(x, y) \end{array}$$

Fuzzy logic has many applications in daily life and science. For example, train control, recognition of handwriting, reduction of fuel consumption, reduction of energy consumption in vacuum cleaners, reasoning with uncertainty, genetic algorithms, etc (see *L. A. Zadeh, et al. 1996 Fuzzy Sets, Fuzzy Logic, Fuzzy Systems, World Scientific Press*).