CS_375/CS_M75 Logic for Computer Science

Ulrich Berger

Department of Computer Science
Swansea University

Fall 2013

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 3rd-year modules offered by my colleagues from the Theory Group, in particular the modules on Interactive Theorem Proving and Embedded Systems. 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] U B, Programming with Abstract Data Types, Lecture Notes, Swansea University, 2009.

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

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]

Let "John is a teacher", "John is rich", and "John is a rock singer" be three atomic propositions. We abbreviate them as A, B, C.

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 formal statement

(*) (A and not(A and B) and (C implies B)) implies not(C)

holds.

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

To prove (*) we assume that the premises A, not(A and B), (C implies B) are all true.

We must show that the conclusion, not(C), is true, that is, C is false. In other words, we must show that assuming C leads to a contradiction.

Hence, we assume C, aiming for a contradiction.

Since we assumed C and we assumed (C implies B) we know that B holds. Hence in fact (A and B) holds since we assumed A. But this contradicts the assumption not(A and B).

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 statement (*) holds: Since we know nothing about the atomic propositions A, B, C, other than that they are either true or false, we can simply try all possible assignments of "true" or "false" to A, B, C and check that statement (*) is true in all cases. It is rather obvious that this method can be automatised as well.

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 expressions

(*) $(A$ and not$(A$ and $B)$ and $(C$ implies $B))$ implies not(C)

(**) not$(A$ and $B)$

are examples of *formulas*, that is, a syntactic representations of a statements. In a given logic it is precisely defined what 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 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*? Looking at the formula (**) we might say that we cannot determine its truth unless we know whether A and B are true. This means that we can determine the truth of (**) 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 with respect to a certain model or 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 a model where a given 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. 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 F is a propositional formula, then $\neg F$ is a propositional formula .

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

Some comments regarding notation:

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 $\neg F$ is called the *negation* of F, pronounced "not F". Some textbooks use the notation $\sim F$ instead.

The formula $(F \wedge G)$ is called *conjunction* of F and G, pronounced "F and G".

The formula $(F \vee G)$ is called *disjunction* of F and G, pronounced "F or G".

A formula of the form $(F \to G)$ is called *implication*, pronounced "F implies G". In many textbooks implication is written $F \supset G$.

It is often convenient to introduce the abbreviation

$$(F \leftrightarrow G) := (F \to G) \wedge (G \to F)$$

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

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

Since in this chapter 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:

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

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 $A \rightarrow B \rightarrow C \rightarrow D$ for $A \rightarrow (B \rightarrow (C \rightarrow D))$.
- Conjunction and disjunction are assumed to associate to the left. For example, we write $A \wedge B \wedge C$ instead of $(A \wedge B) \wedge C$.

Warnings: We do *not* assume that $\wedge$ binds stronger than $\vee$, and, for example, in the formula $(A \to B) \to C$ the parentheses may *not* be omitted. Furthermore, the formulas $\neg A \wedge B$ and $\neg(A \wedge B)$ 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 $(\neg A) \wedge B$.

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 $A \wedge B \to A \vee C \to D$ all omitted parentheses.

(c) Is the expression $A \wedge B \vee C$ 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.

2.2.5 Formulas are inductively generated

The way formulas are defined in 2.2.1 is an example of an *inductive definition*. Instead of going into the (vast) theory of this kind of definition I try to give the intuition (which will be enough for this course). 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 $P(F)$ holds for all formulas F it is enough to prove

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

2.2.6 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

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

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

Induction step: Consider a composite formula, for example, $(F \wedge G)$. By induction hypothesis F and G both satisfy P. Hence clearly $(F \wedge G)$ 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.

2.2.7 Formulas are freely generated

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

2.2.8 Example of a function defined by recursion on formulas

We define the *depth* of a formula as follows

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

We also let $\text{con}(F)$ be the number of occurrences of logical connectives in the formula F.

Lemma 2. $\text{con}(F) < 2^{\text{depth}(F)}$.

Proof. By induction on formulas.

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

Induction step: We have to consider composite formulas.

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

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

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

$$\begin{aligned} \text{con}(F \diamond G) &= \text{con}(F) + \text{con}(G) + 1 \\ &< \text{con}(F) + 1 + \text{con}(G) + 1 \\ &\overset{\text{i.h.}}{\leq} 2^{\text{depth}(F)} + 2^{\text{depth}(G)} \\ &\leq 2 * 2^{\max(\text{depth}(F), \text{depth}(G))} \\ &= 2^{\text{depth}(F \diamond G)} \end{aligned}$$

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]).

2.2.9 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.*

2.2.10 Exercise

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

2.2.11 Exercise

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

2.2.12 Exercise

Compute depth(F) and con(F) for the following formulas F:

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

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

2.2.13 Exercise

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

2.2.14 Exercise

Prove $\text{atom}(F) \leq 2 \cdot \text{con}(F) + 1$.

2.2.15 Exercise

Define a Haskell data type of propositional formulas.

2.2.16 Exercise

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

2.3 Semantics of propositional logic

The semantics (= meaning) of a formula F 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 F depends on the meaning of the atomic propositions occurring in F. Once the meaning of these atomic propositions is fixed, the meaning of F 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:

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

For example,

$$\begin{aligned} \mathrm{H}_\to(0,0) &= 1 \\ \mathrm{H}_\to(0,1) &= 1 \\ \mathrm{H}_\to(1,0) &= 0 \\ \mathrm{H}_\to(1,1) &= 1 \end{aligned}$$

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 $\mathcal{A}$.

2.3.1 Definition (Assignment)

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

2.3.2 Definition (truth value of a formula)

We define the *truth value of F under an assignment α*

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

by recursion on formulas:

$$\begin{aligned} [\![P_i]\!]\alpha &= \alpha(P_i) \\ [\![\top]\!]\alpha &= 1 \\ [\![\bot]\!]\alpha &= 0 \\ [\![\neg F]\!]\alpha &= \mathrm{H}_\neg([\![F]\!]\alpha) \\ [\![F \diamond G]\!]\alpha &= \mathrm{H}_\diamond([\![F]\!]\alpha, [\![G]\!]\alpha) \quad \text{for } \diamond \in \{\wedge, \vee, \to\} \end{aligned}$$

2.3.3 Definition (substitution)

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

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

2.3.4 Exercise

Define $F[A := G]$ by recursion on the formula F.

Solution.

$$\begin{array}{rcl} A[A := G] & = & G \\ B[A := G] & = & B \quad \text{if } B \text{ is an atomic proposition different from } A \\ \top[A := G] & = & \top \\ \bot[A := G] & = & \bot \\ (\neg F)[A := G] & = & \neg(F[A := G]) \\ (F \diamond F')[A := G] & = & F[A := G] \diamond F'[A := G] \quad \text{for } \diamond \in \{\wedge, \vee, \to\} \end{array}$$

2.3.5 Substitution Theorem

Let F, G be formulas and A an atomic proposition. For every assignment α we have

$$[\![F[A := G]]\!]\alpha = [\![F]\!]\alpha'$$

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

Proof. By induction on formulas. We refer to the recursive definition of substitution given in the solution to Exercise 2.3.4.

$[\![A[A := G]]\!]\alpha = [\![G]\!]\alpha = \alpha'(A) = [\![A]\!]\alpha'$

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

$$\begin{aligned}
[\![\neg F[A := G]]\!]\alpha &= [\![\neg F[A := G]]\!]\alpha \quad \text{by definition of substitution} \\
&= \mathrm{H}_\neg([\![F[A := G]]\!]\alpha) \quad \text{by definition of the value of a formula} \\
&= \mathrm{H}_\neg([\![F]\!]\alpha') \quad \text{by induction hypothesis} \\
&= [\![\neg F]\!]\alpha' \quad \text{by definition of the truth value of a formula}
\end{aligned}$$

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

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

$$\begin{aligned}
\alpha[A := b](A) &:= b \\
\alpha[A := b](B) &:= \alpha(B) \quad \text{if } B \text{ is an atomic proposition different from } A
\end{aligned}$$

Remark. The Substitution Theorem says that the value of a formula which contains a formula G as a subformula only depends on the *value* of G 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 F we may replace a subformula G by an equivalent formula G' without changing the meaning of F (this is shown in the Equivalence Theorem 2.3.13 below; what it means for two formulas to be equivalent is defined in 2.3.9).

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.

2.3.6 Definition (logic gate)

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

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

2.3.7 Exercise

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

2.3.8 Definition (logic gate defined by a formula)

Since an assignment α 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 F can also be viewed as a *boolean function* or *logic gate*

$$\mathrm{H}_F : \{0,1\}^n \to \{0,1\}, \quad \mathrm{H}_F(\alpha) := [\![F]\!]\alpha$$

2.3.9 Definition (equivalence)

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

2.3.10 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:

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

2.3.11 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.

2.3.12 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) $\neg A \to B$

2.3.13 Equivalence Theorem

Let F, F', G, G' be formulas and A an atomic proposition.

If $F \equiv F'$ and $G \equiv G'$, then $F[A := G] \equiv F'[A := G']$.

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.

2.3.14 Definition (satisfaction, model, tautology)

If $[\![A]\!]\alpha = 1$ we say α *satisfies* A, or α *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 α.

Remark. Speaking in terms of logic gates, for a formula F to be a tautology means that the logic gate H_F it defines is constant 1.

Similarly, for F to be a unsatisfiable means that the logic gate H_F is constant 0.

In particular, all tautologies are equivalent and all unsatisfiable formulas are equivalent.

Formulas F that are neither tautologies nor unsatisfiable (which is the case for most formulas) define a logic gate H_F which is not constant, that is, there are assignments α, α' such that $[\![F]\!]\alpha = 1$ and $[\![F]\!]\alpha' = 0$.

2.3.15 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) $B \to \neg A$

2.3.16 Exercise

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

2.3.17 Theorem

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

Proof. The proof has two parts.

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

$$\begin{aligned} [\![\neg F]\!]\alpha &= \mathrm{H}_\neg([\![F]\!]\alpha) \quad \text{by definition of the value of a negated formula} \\ &= \mathrm{H}_\neg(1) \quad \text{because } F \text{ is a tautology} \\ &= 0 \end{aligned}$$

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

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

$$\begin{aligned} [\![F]\!]\alpha &= \mathrm{H}_\neg(\mathrm{H}_\neg([\![F]\!]\alpha)) \\ &= \mathrm{H}_\neg([\![\neg F]\!]\alpha) \quad \text{by definition of the value of a negated formula} \\ &= \mathrm{H}_\neg(0) \quad \text{because } \neg F \text{ is unsatisfiable} \\ &= 1 \end{aligned}$$

2.3.18 Theorem (closure of tautologies under arbitrary substitution)

Let F and G be formulas and A an atomic proposition.

$$\text{If } F \text{ is a tautology, then } F[A := G] \text{ is a tautology.}$$

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

2.3.19 Exercise

(a) Does the converse of Theorem 2.3.18 hold? That is, is the following true?

$$\text{If } F[A := G] \text{ is a tautology, then } F \text{ is a tautology.}$$

(b) Does Theorem 2.3.18 still hold if "is a tautology" is replaced by "is satisfiable"?

(c) Does Theorem 2.3.18 still hold if "is a tautology" is replaced by "is unsatisfiable"?

Justify your answers.

2.4 Normal forms

In this chapter we introduce special classes of formulas which are built from a restricted set of logical connectives, but still are rich enough to define all logic gates.

2.4.1 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 $L, L_1, \ldots$.

2.4.2 Definition (literal, conjunctive normal form)

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

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

where each C_i is a disjunction of literals. The formulas C_i are also called (disjunctive) clauses.

2.4.3 Exercise (pigeon hole principle)

We assume that $A_1, A_2, B_1, B_2, C_1, C_2$ are atomic propositions. The following is an example of a formula in CNF:

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

Is this formula satisfiable?

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

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 formula above is in 2-CNF. In general, the pigeon hole formula for n pigeons and m holes is in m-CNF.

Stephen Cook proved in 1971 [1] that the satisfiability problem for formulas in 3-CNF is **NP**-complete. This problem is also known as **3-SAT**. On the other hand, the **2-SAT** problem, that is, 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).

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:

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

Many problems can naturally expressed by Horn Formulas, for example, in logic programming and in expert systems. The satisfiability problem for Horn Formulas can be solved in polynomial time, in fact it is **P**-complete.

[1] Stephen A. Cook, The Complexity of Theorem-Proving Procedures, Proc. 3rd ACM Symp. Theory of Computing, May 1971

2.4.4 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 conjunction of literals. The formulas D_i are also called conjunctive clauses.

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

2.4.5 Exercise (satisfiability for DNF)

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

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

2.4.6 Theorem (completeness of CNF and DNF)

For every logic gate there exists a formula in CNF and also a formula in DNF defining it.

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 α. Recall that we identify the set of assignments with the set $\{0,1\}^n$ of n-tuples of Boolean values by identifying an assignment α with the tuple $(\alpha(P_1), \ldots, \alpha(P_n)) \in \{0,1\}^n$.

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

We encode each α_i by a conjunctive clause D_i as follows:

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

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 α and any $j \in \{1, \ldots, n\}$

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

Therefore,

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

which means that D_i does indeed properly encode α_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 α be an assignment. We have to show $[\![F]\!]\alpha = g(\alpha)$, that is, $[\![F]\!]\alpha = 1 \Leftrightarrow g(\alpha) = 1$.

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

2.4.7 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

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

In general, we call a class of formulas complete if every logic gate can be defined by a formula in the class. Theorem 2.4.6 shows that DNF and CNF are complete.

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.

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

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

2.4.9 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.

2.4.10 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 α_1 and α_2 are models of F (that is, $[\![F]\!]\alpha_1 = [\![F]\!]\alpha_1 = 1$), then the assignment α, defined by $\alpha(A) := \mathrm{H}_\wedge(\alpha_1(A), \alpha_2(A))$, is also a model of F. Conclude from this that the logic gate $\mathrm{H}_\vee$ 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 2.4.5, 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.

2.4.11 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

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

2.5 The Compactness Theorem

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

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 $\alpha = (\alpha(P_1), \alpha(P_2), \ldots, \alpha(P_n)) \in \{0,1\}^n$,

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

2.5.1 Definition (Model, Satisfiability, for a set of formulas)

Let Γ be a set of formulas.

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

$$\alpha \models \Gamma$$

Γ is *satisfiable* if it has a model.

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

2.5.2 Definition (Logical consequence)

A formula G is a *logical consequence* of a set of formulas Γ if every model of Γ is a model of G, that is, for all assignments α, if $\alpha \models \Gamma$, then $[\![G]\!]\alpha$. One also says Γ *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:

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

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

2.5.3 Exercise

Let Γ 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 Γ is satisfiable, then G is satisfiable.

2.5.4 Exercises (Schoening, Ex. 7)

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

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

2.5.5 Definition (Binary Tree)

Let $\beta \in \{0,1\}^n$ and $\beta' \in \{0,1\}^m$ be finite assignments. We say β' is a *restriction* of β, or β is an *extension* of β', written $\beta' \subseteq \beta$, if $m \leq n$ 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.

2.5.6 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, α is an infinite path in T. This completes the proof.

*R*emarks.

1. The construction of the infinite path α 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.

2.5.7 Theorem (Compactness Theorem)

Let Γ be a set of formulas such that every finite subset of Γ is satisfiable. Then Γ is satisfiable.

Proof. If there are only finitely many atomic formulas occurring in Γ, then the number of logic gates defined by some formula in Γ is finite. 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 α which necessarily is then a model of Γ.

Hence, we consider now the interesting case that Γ is infinite, say

$$\Gamma = \{F_1, F_2, \ldots\}$$

and there are infinitely many atomic propositions occurring in Γ, say

$$P_1, P_2, \ldots$$

We set $G_n := F_1 \wedge \ldots \wedge F_n$. By assumption, each of the formulas G_n is satisfiable.

Below, we let β, β' range over finite assignments. We define a binary tree T as follows:

$$T := \{\beta \mid \exists n \, \exists \beta' \, (\beta \subseteq \beta' \wedge \beta' \models G_n)\}$$

T is infinite, since every model of some G_n is in T. By König's Lemma, T has an infinite path α. In order to show that α is a model of Γ it suffices to show that α satisfies every G_n. Set $\beta := \overline{\alpha}(m)$ where $m \geq n$ is so large that all atomic propositions occurring in G_n are among $\{P_1, \ldots, P_m\}$. Since α is a path in T, we have $\beta \in T$, which means that there is some $k \geq m$ and some extension β' of β with $\beta' \models G_k$. It follows $\beta \models G_n$ and therefore $\alpha \models G_n$. This completes the proof.

2.5.8 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 Γ_0 of Γ.

Proof. This is left as an easy exercise.

2.5.9 Definition (Axiom system)

Let Γ be a set of formulas. A set of formulas Γ_0 is an *axiom system* for Γ if Γ and Γ_0 have the same models, that is for every assignment α,

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

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

2.5.10 Exercise

Show that Γ_0 is an axiom system for Γ if and only if (i) $\Gamma \models G$ for every $G \in \Gamma_0$ and (ii) $\Gamma_0 \models F$ for every $F \in \Gamma$.

2.5.11 Exercise

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