Logic in Computer Science

—Lecture Notes for COM2003/364—

Georg Struth
`g.struth@sheffield.ac.uk`

18.02.2018

Contents

Preface

This booklet contains the lecture notes for the course COM2003/364 on *Logics in Computer Science*, taught at the University of Sheffield in Spring 2017/18. The notes are work in progress and may contain typos and errors. They will be completed and updated during the semester. I am very grateful for any comments and suggestions for corrections. Please email them to my University of Sheffield address.

The notes are formatted specifically for electronic display, including hyperlinks. Please safe some trees and avoid printing out paper copies. The notes are provided free of charge for personal use only; they should not be distributed.

Other sources are not adequately acknowledged or cited in this text so far. Michael Huth and Mark Ryan's book on *Logic in Computer Science* and Dirk van Dalen's book on *Logic and Structure* have been helpful for compiling this material.

Last changes: 18.02.2018

Georg Struth
Sheffield in Spring 2018

Chapter 1

Introduction

1.1 Organisation and Learning Guide

This course, COM2003/364, is an introduction to logic for computer scientists at undergraduate level. It assumes little beyond secondary school mathematics and basic propositional logic. Notions and notations from discrete mathematics, functions and relations, sets and trees, recursion and induction, as taught in COM1002, are helpful for later lectures. Those who did not attend COM1002 may find a gentle introduction in my textbook on *Modelling Computing Systems, Mathematics for Computer Science*, co-authored with Faron Moller.

COM2003/364 is organised into two hours of lectures and a one-hour exercise session each week. It is assessed by exam (2h) at the end of the term.

I do not use use MOLE or similar university resources. Instead I supply all relevant material and information online at my course web site

`https://staffwww.dcs.shef.ac.uk/people/G.Struth/campus_only/logic`

Please check it on a regular basis.

I neither use slides nor recordings for these lectures. I prefer to teach jurassic-style at the blackboard and have written these lecture notes instead, which contain everything you need to know. My main aim is to teach you mathematical thinking and using logic for rigorous modelling and reasoning, especially about computing systems. I also use state-of-the-art computer software for programming and executing logical proofs in class. The blackboard is just perfect for this style of teaching, and it is essential that it happens spontaneously and interactively in front of you.

The exercise sessions complement my lectures in important ways. These are held in small groups by a demonstrator. At the beginning of each week I publish an exercise sheet at the course web site. You are asked to prepare and present solutions during the exercise sessions at the end of each week. The exercises are not marked—I do not have the resources for this—but working through all sheets, spending time to write them down carefully, and participating actively in the exercise sessions is as least as helpful as attending my lectures.

First of all, you don't understand logic unless you can do the exercises, and they are the best exam preparation. Secondly, the sessions give you essential feedback: whether you have understood a topic, where you have gaps, and whether your solutions are to the standards expected. Finally, they provide great opportunities for asking questions and discussing logic with others.

Logic is quite different from mathematics at school—not very difficult, but learning it requires time, persistance and hard work. A busy semester schedule should not prevent you from putting some work in week by week. Little gaps become big gaps rather quickly.

These lecture notes are quite self-contained. They cover the expected learning outcomes of this course and everything that is relevant to the exam. A less formal introduction to logic can be found in my aforementioned textbook. It explains in particular the basic syntax of propositional and predicate logic and contains numerous examples for translating natural language sentences, for recursive definitions and for proofs by induction. For further reading I recommend Michael Huth and Mark Ryan's book on *Logic in Computer Science* and Dirk van Dalen's book on *Logic and Structure*. Beyond that you might look out for any other book with the word *logic* in the title.

1.2 Why Study Logic?

Logic is an old subject—by far the oldest one you will encounter studying computer science. It has been around for thousands of years. Aristotle was one of its founding fathers in Ancient Greece. It became part of mathematics at the end of the nineteenth century, helped giving birth to the notion of computation almost hundred years ago through work of Gödel, Church, Turing and others, and has been a core topic of the emerging field of computer science since the second half of the twentieth century. Since then, many developments in logic were driven by computing, and many advances in computing were based on logic. No other area of computer science has received as many Turing Awards.

Today, logic, from foundations to applications, plays a fundamental role in our field.

- It is an important part of computability theory—the core of computing in a very narrow sense—and the theory of NP-completeness.

- It is essential to program semantics; the study of the meaning of programs as mathematical objects. Logics of programs, in particular, yield precise descriptions of the executions of programs and the relationships between them.

- It provides a foundation to type theory; there is a fundamental correspondence between (functional) programs and proofs in certain logical formalisms.

- Logical queries over finite mathematical structures and their complexity are important to the foundations of data bases, the design of hardware circuits or the verification of concurrent reactive systems.

- Logics are important for modelling and analysing the correctness of programs, hardware and software systems, e.g. security protocols, knowledge and beliefs among agents, or ontologies or knowledge in expert systems.

- The design of interactive, automated theorem provers, model checkers and other solvers, based on various logics, supports many of the activities mentioned.

- Last, but not least, logic is a fundamental part of circuit design.

It is therefore rather unsurprising that logic is a core part of the computer science curriculum at universities around the world. But beyond that, by separating valid from fallacious reasoning and emphasising a formal treatment of language, logic is an essential part of the scientific method. It is therefore relevant to a wide range of academic subjects beyond computer science and mathematics, from philosophy to law.

Finally, work by Hilbert, Gödel, Tarki, Turing and Church on the potential and limitations of logic and the nature of computation belong, together with quantum mechanics and relativity theory, to the greatest and most influential intellectual achievements of the twentieth century. This alone makes logic well worth studying.

1.3 Course Overview

These lectures are devided roughly into three parts. The first one, Chapters 2 and 3, covers propositional and predicate logic. Its main purpose is to introduce logic as a universal tool for rational reasoning; a cornerstone of the scientific method. Beyond this rather foundational philosophical view, logic is developed as a mathematical subject with precise definitions, theorems and proofs, I outline some deep connections between logic and computation and explain how logics and logical reasoning can be programmed and executed on a machine.

The second part, Chapter 4, is devoted to automated deduction: the implementation of automated proof search algorithms on machines. Proof search is an intractable computational task—something we cannot expect to perform successfully on a machine. Implementations of automated proof search in SAT and SMT solvers and in first-order automated theorem provers are nevertheless widely and successfully used in the hardware and software industry, and increasingly integrated into industrial development processes. Beyond computer science, they have been used for proving some hard mathematical theorems. This part presents the basic proof search algorithms and procedures, and thus gives a glimpse at the anatomy of contemporary automated deduction technology.

The third part of these lectures, Chapter 5, introduces temporal logics, a formalism with important applications in model checking, a method for verifying the correctness of hardware, software and control systems automatically with computers. This approach is increasingly used in industrial practice as well. I cover the basics of the approach and show you some simple examples with a state-of-the-art model checking tool.

The lectures thus proceed from foundations to applications. They also prepare for more advanced modules in later years, in particular COM4507 Hardware and Software Verification, COM3501 Computer Security and Forensics and COM3190 Theory of Distributed Systems.

Chapter 2

Propositional Logic

2.1 What is Logic?

Logic, like any other science, deals with evidence and truth. Yet unlike any other science, it does not aim at discovering new facts, but at inferring or deriving new facts from old ones; at making judgments about their relationship. Logic is therefore often called the science of reasoning, the study of the laws of thought. It thus deals with arguments and their validity, with rational discourse, rather than with particles, molecules, plants, ecosystems or human behaviour. To computer scientists, more specifically, it offers precise formal languages, rigorous notions of meaning, evidence and truth, and powerful tools for proving facts from hypotheses, often algorithmically and automatically.

Logic does not even deal with concrete arguments, judgments or proofs, but merely with their form—with abstract universal patters. It can therefore be described as *symbolic.* Let me try to give an example.

Example 2.1. The sentences

> *All humans are mortal. Socrates is human.*
> *Therefore, Socrates is mortal.*

certainly form a valid judgment: we have de facto evidence that Socrates was human, and that all humans are mortal. This gives us sufficient evidence to accept the conclusion that Socrates was mortal. This evidence is hard to grasp and may seem rather mysterious. It is an abstract rule of thought rather than a concrete mechanism that controls traffic lights or explains chemical reactions. It is obtained by applying a rule of logic. If we are prepared to accept the premises of the judgment about Socrates, then logic tells us that we should accept its conclusion as well.

Symbolic logic allows us to rewrite this judgment as

$$\frac{\forall x.\ \mathit{Human}(x) \to \mathit{Mortal}(x) \qquad \mathit{Human}(\mathit{Socrates})}{\mathit{Mortal}(\mathit{Socrates})}$$

and even more symbolically as

$$\frac{\forall x.\ H(x) \to M(x) \qquad H(s)}{M(s)}$$

abbreviating subjects and predicates in the obvious way. Once we are left with this symbolic representation, it turns out that the abstract form of this judgment is all that matters: its form alone makes us accept this judgment as valid, not its particular content. This was one of the great insights of logicians two thousand years ago, and it led to the search for universal laws of reasoning. If we are prepared to accept such universal laws as rules of the game of logic, and if we agree with the evidence provided for the hypotheses of a particular argument, then we are committed to accepting the conclusions derived in an essentially algorithmic fashion, and we can check the validity of arguments like we can verify the moves of a game of chess.

As H, M, x and s are merely symbols, part of the syntax, in the parlance of logic, we can interpret them, give them a meaning or semantics, in various ways and replace them by other symbols. We may, for instance, write

$$\frac{\forall x.\ P(x) \to Q(x) \qquad P(a)}{Q(a)}$$

and the judgment remains the same. We may also interpret symbols differently, for instance, write

All birds can fly. Tweety, the penguin, is a bird.
Therefore, Tweety can fly.

and the judgment is still valid. In fact this is crucial for understanding the role of logic.

Judgments are valid irrespective of the truth or falsity of particular premises and conclusions—logic has to be oblivious to particular facts if its rules are to be universal. Tweety not only provides counterevidence to the conclusion, but also to the first premise of the judgment. Logic, of course, cannot be held responsible if we derive particular false conclusions from particular hypotheses that are not true. □

Symbolic logic thus distinguishes clearly between the form of judgments and factual evidence. Over the centuries, logicians have reduced complex patterns of reasoning to a small set of simple, independent and rather intuitive rules. These can be applied in unambiguous and rigorous formal ways to sentences presented in formal symbolic languages. Ultimately, logics can therefore be implemented and executed on a machine.

The ultimate question about the origin or justification of the rules of logic is certainly interesting, but can be left to philosophers and historians. We may content ourselves with the fact that they are widely accepted by mathematicians and valuable for a wide range of computer science applications. In such applications, logic is often too complex and sometimes too important to be left to humans. Logical analyses of safety or security critical applications, where systems must be examined at high levels of rigour to prevent failure, should by any means be performed on a machine. A discussion of some high-profile engineering failures can be found in my book on *Modelling Computing Systems, Mathematics for Computer Science.* Here I only present two little examples to point out that we may not be as good at logical reasoning as we think.

Example 2.2. Consider the following line from an old song:

Everybody loves my baby, but my baby don't love nobody but me.

Who would have thought that it follows logically that *I am my baby*? □

Example 2.3 (Wason's selection test)**.** Four cards on a table display the symbols 5, 8, A and B, respectively. Which one(s) do you need to flip to verify the claim that *if one side of card shows an even number, then its other side shows an A*? The vast majority of participants reportedly fail this simple test (you need to flip 8 and B). The test is related to the meaning of implication in propositional logic. Hence a vast majority of people finds it difficult to reason about implication. □

In sum, logic is an old subject, and for computer scientists an important one. We all use logic in one way or another when we reason. Mathematicians and other scientists use it heavily for presenting and structuring their arguments and proofs; they have often learned the rules of the game implicitly through practice and experience. Computer scientists and engineers use logics explicitly as formal languages for specifying and reasoning about the systems they build. This course presents logic from all three points of view.

2.2 Propositions and Judgments

This section takes a rather foundational view on logic, with little emphasis on formality. It motivates some basic inference rules with as little formal baggage as possible, as patterns of reasoning or proof strategies.

Propositional logic is based on statements that are called *propositions*, and here is where we begin. We should think of propositions simply as declarative sentences that express some facts about the world, and can therefore be true or false. Sentences that make factual claims about abstract concepts or things that exist only in our minds, such as

3 *is a magic number.* or *Pegasus is a winged stallion.*

may be included. A claim such that

φ *is a proposition.*

is called a *judgment*. Another kind of judgment we care about is

φ *is true.*

What judgments are, is an old philosophical question. We may see a judgment as an act of knowing something, in the sense that we can provide and are prepared to accept evidence for it.

For arguments and logical reasoning, *hypothetical judgments* are particularly important. These are of the form

J from hypotheses $J_1, \ldots, J_n$.

We accept such a judgment if we are prepared to accept J when assuming evidence for the hypotheses or assumptions, $J_1, \ldots, J_n$, hence if we suppose there is evidence that the hypotheses $J_1, \ldots, J_n$ support the conclusion J. The judgment in the conclusion thus depends on the judgments made about the hypotheses. In a hypothetical judgment, the evidence for the hypotheses remains implicit, We do not need to prove hypotheses proof, although often we could. In discussions we are often challenged to defend our assumptions, and often this does not lead anywhere. Discussions always need some common ground, some basic hypotheses that are accepted and not questioned any further.

The process by wich we give explicit evidence for a judgment is called a *proof*, *deduction* or *derivation*. We use these notions as synonyms. We write

$$\begin{array}{c} J' \\ \vdots \\ J \end{array}$$

to indicate a deduction with hypothesis J' and conclusion J, or simply

$$\begin{array}{c} \vdots \\ J \end{array}$$

for a deduction with conclusion J.

2.3 Natural Deduction

Next we present the rules of the game: the rules of *natural deduction* for propositional logic—our basic tool for building logical arguments and proofs. Natural deduction was proposed by Gerhard Gentzen and Stanisław Jaśkowski around 1936 with the aim of providing a set of basic rules that come as close to the natural way we reason as possible. In particular, the rules of natural deduction are supposed to reflect mathematical practice. We introduce these rules semi-formally to do their naturalness justice.

We use a basic set $\varphi, \psi, \chi \ldots$ of *schematic variables* for propositions in judgments. Remember that variables are placeholders for values: we can instantiate them by values. Schematic variables take propositions as values, and nothing else. Being more pedantic, we should require a *countable* or *countably infinite* set of variables, that is, as many of them as there are natural numbers.

The two judgments we care about are of the form φ *is a proposition* and φ *is true*, which we abbreviate as

φ *prop* and φ *true*.

The following *structural rules* of natural deduction are needed for managing hypotheses of the form φ *true* in derivations.

- Each hypothesis may be used as a conclusion.

- Hypotheses need not be used in a derivation.
- Each hypothesis can be used several times in a derivation.
- Hypotheses can be used in any order in a derivation.

The use of these structural rules will become clear by example as we go along.

Next we turn to the particular inference rules of natural deduction for propositional logic, the basic building blocks of derivations. These describe the use of the *propositional connectives* of propositional logic. Namely $\wedge$, the symbol for conjunction (and), $\to$, the symbol for implication (if ... then), $\vee$, the symbol for disjunction (or), $\neg$, the symbol for negation (not), $\top$, the symbol for true and $\bot$, the symbol for false.

We freely add brackets for disambiguation, as in high school arithmetic. We introduce the rules of natural deduction connective by connective, presenting examples of their use in Section 2.4. A complete list of rules can be found in Section 2.11.

2.3.1 Conjunction

We start with the rules for conjunction. First of all we want to express the judgment that if p and q are propositions, then so is $\varphi \wedge \psi$. We write this as

$$\frac{\varphi\ \mathit{prop} \qquad \psi\ \mathit{prop}}{\varphi \wedge \psi\ \mathit{prop}}\ \wedge F$$

where the label $(\wedge F)$ indicates that this is a *formation rule* for propositions (as opposed to an inference rule). Applying this rule iteratively, we can derive $(\varphi \wedge \psi) \wedge \chi$ *prop* from the hypotheses φ *prop*, ψ *prop* and χ *prop*. We can represent the resulting deduction by the tree

$$\frac{\dfrac{\varphi\ \mathit{prop} \qquad \psi\ \mathit{prop}}{\varphi \wedge \psi\ \mathit{prop}} \qquad \chi\ \mathit{prop}}{(\varphi \wedge \psi) \wedge \chi\ \mathit{prop}}$$

The construction of larger trees follow the same pattern.

Next we introduce the *inference rules* for conjunction. These describe the use of conjunction in derivations; hence relate to judgments about truth and the operational meaning of conjunction. There are several ways of motivating these rules.

One way, which is pursued in *constructive* or *intuitionistic* logic, identifies the meaning of a connective with the methods of proving it. It asks when we have evidence for a proof of $\varphi \wedge \psi$. Obviously, this is the case if we have a proof of φ and a proof of ψ, hence a pair of proofs for φ and ψ. From a computational point of view, we could even ask when we have a program that achieves task $\varphi \wedge \psi$—which is the case if we can supply a pair of programs (which we may run in parallel), one of which achieves φ, and other one ψ.

An alternative way, which is pursued in *classical logic*, is to identify the meaning of a proposition simply with its *truth value*: true or false. We then ask for conditions under which $\varphi \wedge \psi$ is true. Obviously, this is the case if and only if φ is true and ψ is true.

Henceforth we follow by and large the classical way and consider judgments of the form φ *true*. Nevertheless we keep the intuitionistic alternative in mind because of its appealing computational content.

Returning to the inference rules of natural deduction, we obtain the following rule for conjunction as a summary of the previous discussion:

$$\frac{\varphi\ \mathit{true} \qquad \psi\ \mathit{true}}{\varphi \wedge \psi\ \mathit{true}}\wedge I$$

We take it as the operational definition of the meaning of conjunction (by contrast to its truth table). Here we tacitly assume that φ and ψ are both propositions (whence $\varphi \wedge \psi$ is a proposition as well). The label $\wedge I$ indicates that the rule is an *introduction rule*, since the conjunction is introduced in its conclusion.

More generally, in any inference rule, we refer to the hypotheses used as *premises*.

Apart from the introduction rule, natural deduction supplies two additional inference rules for conjunction. On the one hand, these tell us what we may conclude from a judgment $\varphi \wedge \psi$ *true* in a proof. On the other hand, they balance the effect of the conjunction introduction rule, as we will see. The two rules are

$$\frac{\varphi \wedge \psi\ \mathit{true}}{\varphi\ \mathit{true}}\wedge E_l \qquad \frac{\varphi \wedge \psi\ \mathit{true}}{\psi\ \mathit{true}}\wedge E_r$$

For obvious reasons, they are called *conjunction elimination* rules. To explain them, we should ask what a hypothesis $\varphi \wedge \psi$ *true* gives us. To answer this question we need to look at the conjunction introduction rule, which defines its meaning. It tells us that we must know both φ *true* and ψ *true*; alternatively that we have a pair of proofs for φ and ψ or even two programs that compute the outputs φ and ψ. Hence that is exactly what we may conclude from $\varphi \wedge \psi$ *true*: that φ *true* and that ψ *true*.

The argument motivating ($\wedge E_l$) can be presented more explicitly s follows. If we expand the premise of the elimination step in the following proof by the corresponding introduction step which explains the conjunction, we obtain a conjunction introduction followed by an elimination in any suitable derivation:

$$\frac{\dfrac{\begin{matrix}\vdots\\ \varphi\ \mathit{true}\end{matrix} \qquad \begin{matrix}\vdots\\ \psi\ \mathit{true}\end{matrix}}{\varphi \wedge \psi\ \mathit{true}}\wedge I}{\varphi\ \mathit{true}}\wedge E_l$$

It is easy to see that this combination was really unnecessary: the same conclusion could have been achieved simply by

$$\begin{matrix}\vdots\\ \varphi\ \mathit{true}\end{matrix}$$

which is the left hypothesis in the derivation! Hence we can always reduce the first kind of derivation to the second one. A similar result can been obtained by using $\wedge E_r$ instead of $\wedge E_l$ with respect to a derivation of ψ *true*. Together these arguments show that conjunction elimination is not too aggressive: it follows directly from the meaning of conjunction, as prescribed by the introduction rule.

Analogously, we can always eliminate a conjunction and then reintroduce it. Every derivation

$$\begin{array}{c} \vdots \\ \varphi \wedge \psi \ \mathit{true} \end{array}$$

with conclusion $\varphi \wedge \psi$ *true* can be expanded to one of the form

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \varphi \wedge \psi \ \mathit{true}\end{array}}{\varphi \ \mathit{true}}\wedge E_l \qquad \dfrac{\begin{array}{c}\vdots\\ \varphi \wedge \psi \ \mathit{true}\end{array}}{\psi \ \mathit{true}}\wedge E_r}{\varphi \wedge \psi \ \mathit{true}}\wedge I$$

This shows that conjunction elimination is not too week either: whenever we eliminate a conjunction somewhere in a derivation, we can recover it immediately if we wish.

On the one hand, each proof state of a conjunction introduction or elimination rule can be recovered by using the two kinds of rules, nothing is gained and nothing is lost. Information is only arranged in a different way—the rules are in perfect balance and satisfy a *principle of harmony*. On the other hand, both patterns, the elimination of a conjunction immediately after its introduction and the introduction of a conjunction immediately after an elimination, can be considered as unnecessary detours in derivations, in particular the former ones can always be avoided.

The discussion in this paragraph conveys three general points about natural deduction:

- Each propositional connective comes with formation rules for building complex propositions from more basic ones, and inferences rules for reasoning with this connective.
- There are two kinds of inference rule: Introduction rules define the operational meaning of a connective. Elimination rules allow us to use connectives in premises of other inference rules.
- Introduction and elimination rules are in harmony. This guarantees that the rules can prove precisely what their meaning allows.

The insight that introducing a conclusion with a certain connective, and then eliminating it again, is redundant and therefore unnecessary gives rise to the notion of *proof normalisation*. It can be shown that all derivations by natural deduction can be rearranged into two phases, a first one eliminating all connectives from hypotheses and a second one introducing connectives to build the conclusion. Derivations of this shape are in *normal form*. This aspect of natural deduction is discussed further in the following section.

2.3.2 Implication

Next we consider implication. To simplify the discussion, we ignore the formation rule; it has precisely the same shape as that for conjunction. In addition, we simply write φ for the judgment φ *true*. Natural deduction once more supplies an introduction and an elimination rule for implication.

$$\dfrac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \to \psi} \to I \qquad\qquad \dfrac{\varphi \to \psi \qquad \varphi}{\psi} \to E$$

The *implication introduction* rule is a hypothetical judgment. If we assume that φ is true and succeed in deriving that ψ is true from this hypothesis, then we can summarise this derivation as the judgment that $\varphi \to \psi$ is true. In other words, $\varphi \to \psi$ is true if we can derive ψ from the hypothesis φ. In the judgment $\varphi \to \psi$, the hypothesis φ is no longer needed; it is internalised into $\varphi \to \psi$—and that is all that implication does! We can therefore *discharge* or *kill* the hypothesis, as indicated by the notation $[\varphi]$. A discharged hypothesis is local to the derivation in which it is used, in the same way variables may be local to a procedure or block of code. However, this does not prevent us from using φ as a hypothesis elsewhere in the derivation, we can simply copy it—as an alive hypothesis—into the derivation.

Alternatively, we can read $\varphi \ldots \psi$ as a method that transforms proof of φ into a proof of ψ, or a program that takes task φ as an input and yields task ψ as an output. The notation $\varphi \to \psi$ then ressembles that of a function type. Once again, the introduction rule defines the operational meaning of implication.

The *implication elimination* rule describes the use of implications in derivations. Once more we should ask what a the premise $\varphi \to \psi$ gives us, and the answer lies in the introduction rule. We know that $\varphi \to \psi$ is true whenever, if φ holds, then ψ must be true as well. Hence if we supply the premise, that φ is true as the second assumption, then ψ is true unconditionally. The elimination rule is so well known that it has a name: *modus ponens*.

Alternatively, if we have a proof of φ and a method for transforming a proof of φ into a proof of ψ, which exists according to the introduction rule, then that gives us a proof of ψ that does no longer depend on the hypothesis φ; and if we supply an input φ to a program that transforms tasks φ into tasks ψ, then we can obtain the output ψ.

To establish harmony, we reduce

$$\dfrac{\dfrac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \to \psi} \to I \qquad \begin{array}{c}\vdots\\ \varphi\end{array}}{\psi} \to E \qquad\qquad \text{to} \qquad\qquad \begin{array}{c}\vdots\\ \varphi\\ \vdots\\ \psi\end{array}$$

and expand

$$\begin{array}{c}\vdots\\ \varphi \to \psi\end{array} \qquad \text{to} \qquad \cfrac{\cfrac{\begin{array}{c}\vdots\\ \varphi \to \psi\end{array} \qquad [\varphi]}{\psi} \to E}{\varphi \to \psi} \to I$$

The reduction formalises once again the explanation of the elimination rule.

2.3.3 Truth and Falsity

There is only an introduction rule for truth and only an elimination rule for falsity.

$$\frac{}{\top}\,\top I \qquad\qquad \frac{\bot}{\varphi}\,\bot E$$

To know that $\top$ is true does not require any assumption and does not allow us to conclude anything else. We may introduce $\top$ without any premise, but not eliminate it. To know that $\bot$ is true, by contrast, requires too much, that is, inconsistent knowledge, and because of that it allows us to derive anything. We may eliminate $\bot$, but not introduce it from any hypotheses that we deem true. In fact, we may safely conclude any φ from $\bot$ just because we may never introduce $\bot$ anyway.

Despite of the symmetry in the rules for $\bot$and $\top$, only $\bot$ is needed. We can define

$$\top = \varphi \to \varphi$$

for any φ. In fact,

$$\frac{[\varphi]}{\varphi \to \varphi}$$

because one of the structural rules tells us that if we can assume φ, then we can also conclude it. The assumption $[\varphi]$ can then be forgotten and we obtain the introduction rule for $\top$.

2.3.4 Disjunction

The introduction rules for $\vee$ are straightforward.

$$\frac{\varphi}{\varphi \vee \psi}\,\vee I_l \qquad\qquad \frac{\psi}{\varphi \vee \psi}\,\vee I_r$$

They hold because $\varphi \vee \psi$ is true if and only if at least one of φ and ψ is. Alternatively, every proof of φ is a proof of $\varphi \vee \psi$ and so is ψ; and every program that can compute either φ or ψ can compute $\varphi \vee \psi$. As previously, this rule defines the operational meaning of disjunction.

By contrast, the elimination rule for disjunction is rather complicated.

$$\frac{\varphi \vee \psi \qquad \begin{array}{c}[\varphi]\\ \vdots\\ \chi\end{array} \qquad \begin{array}{c}[\psi]\\ \vdots\\ \chi\end{array}}{\chi}\,\vee E$$

If we know that $\varphi \vee \psi$ is true, then we have a proof of either φ or ψ, by the introduction rule. In the first case, we can add evidence for φ to the hypothetical proof of χ, which yields a proof of χ that no longer depends on φ. In the second one we can add evidence for ψ in the hypothetical proof of χ, which yields a proof of χ that no longer depends on ψ. Hence in any case there is a proof of χ which no longer depends on the hypotheses φ or ψ.

Disjunction elimination is nothing but the principle of *case analysis*. For proving χ from the hypothesis $\varphi \vee \psi$, it suffices, by case analysis, to derive χ from φ and separately from ψ. If this succeeds we know that χ can be derived from $\varphi \vee \psi$ and the particular hypotheses φ and ψ in the case analysis can be killed. This is illustrated by the following simple example.

Example 2.4. We prove by case analysis on n that, for all numbers n, the term $n^2/2$ has either remainder 1 or 0.

- If n is even, then $n^2 = 4k^2$ for some number k; division by 2 has remainder 0.
- If n is odd, then $n^2 = 4(k^2 + k) + 1$ for some number k; division by 2 has remainder 1.

Thus in either case the claim is true. □

As usual, it is possible to prove harmony between the introduction and elimination rules, but we do not spell out the details. Instead we argue why an alternative elimination rule

$$\frac{\varphi \vee \psi}{\varphi}\,?$$

would fail. The derivation

$$\cfrac{\cfrac{\cfrac{}{\top}\,\top I}{\varphi \vee \top}\,\vee I_r}{\varphi}\,?$$

shows that we can derive any φ in the presence of this rule, hence in particular $\bot$. This destroys harmony and introduces inconsistency. Intuitively, we have evidence for $\varphi \vee \psi$ if and only if we have evidence for one of φ, ψ, but not for a particular one of the two.

2.3.5 Negation

The natural deduction rules introduced so far have a pleasant modularity property. They spoke for themselves and did not mention the other propositional connectives. The operational meaning of each propositional connective was thus clearly defined in isolation.

It is possible to give such inference rules for negation as well, but these are slightly contrived. Instead we define negation explicitly with the help of the other symbols. Obviously, the negation $\neg\varphi$ of φ is true if and only if φ is false. In that case we should be able to derive $\bot$ from the hypothesis φ, and therefore $\varphi \to \bot$ by $\to I$. Hence we take

$$\neg\varphi = \varphi \to \bot$$

as the definition of negation: "*If φ is true, then I'll eat my hat*".

From this definition we can now derive an introduction and an elimination rule for negation simply by instantiating the corresponding rules for implication.

$$\dfrac{\begin{matrix}[\varphi]\\ \vdots\\ \bot\end{matrix}}{\neg\varphi}\,\neg I \qquad\qquad \dfrac{\varphi \qquad \neg\varphi}{\bot}\,\neg E$$

Instantiation means that $\varphi \to \bot$ has been replaced by $\neg\varphi$ in the implication rules. Though these two rules are not strictly needed, they are very natural and often helpful in proofs. The introduction rule for negation tells us that we can conclude $\neg\varphi$ provided we can derive something inconsistent from the hypothesis φ. The elimination rule simply says that φ and $\neg\varphi$ cannot be true at the same time (assuming that both is the case is inconsistent). Harmony of these two rules follows immediately from that of implication.

In sum, we have now derived inference rules for the most important connectives of propositional logic: conjunction, implication, truth, falsity, disjunction, negation. We have justified each inference rule either by appealing to truth conditions, or with a view to proofs or even programs. The latter give the rules computational content. The computational interpretation has rather been a metaphor, but it is very important for type theory in programming. Due to this, the fragment of natural deduction introduced so far is called *constructive* or *intuitionistic*. We will add further rules to this fragment to obtain classical propositional logic in Section 2.6—and loose the nice symmetries and computational content.

2.4 Natural Deduction at Work

This section provides some examples that show how the rules of natural deduction—its intuitionistic fragment, to be precise—can be used for building derivations. The examples are rather artificial. They all used schematic variables instead of real-world sentences or propositions. Yet, by analogy, it certainly makes sense to look at particular features of chess on a board before starting to play games. Interestingly, concrete examples for the informal use of natural deduction can be found in later sections, such as Section 2.9, when we prove properties of propositional logic.

We have already sneaked some little derivations into the previous section, and these suggest that derivations can be associated with trees. The root of a derivation tree is labelled with the conclusion obtained, the leaves are labelled with the hypotheses used. The inner nodes are labelled with the premises and conclusions of intermediary applications of inference rules.

Example 2.5. Suppose we wish to prove $\varphi \to \chi$ from the hypotheses $\varphi \to \psi$ and $\psi \to \chi$.

Starting with the proof goal, we realise that we can potentially obtain a proof by $(\to I)$. This requites us to derive χ from the hypothesis φ, and then kill φ.

$$\dfrac{\begin{matrix}[\varphi]\\ \vdots\\ \chi\end{matrix}}{\varphi \to \chi}\to I$$

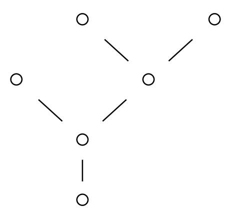
Now we can feed φ into the hypothesis $\varphi \to \psi$ and derive ψ with $(\to E)$.

$$\dfrac{\varphi \to \chi \qquad [\varphi]}{\psi} \to E$$
$$\vdots$$
$$\dfrac{\chi}{\varphi \to \chi} \to I$$

Next we can feed ψ into the hypothesis $\psi \to \chi$ and derive χ with $(\to E)$, as desired.

$$\dfrac{\dfrac{\psi \to \chi \qquad \dfrac{\varphi \to \chi \qquad [\varphi]}{\psi} \to E}{\chi} \to E}{\varphi \to \chi} \to I$$

This completes the derivation; the following tree structure has emerged.



$\square$

This example reveals a general pattern which we discussed briefly in Section 2.3.1. We built the derivation by working from both ends of the proof tree; backwards from the proof goal as well as forwards from the hypotheses. In particular, we closed the gap in the middle of the proof moving upwards towards the hypotheses with introduction rules and downwards towards the proof goal with elimination rules.

This is the general work flow of natural deduction. It is enabled by the normal form property of natural deduction proofs mentioned in Section 2.3.1. The proof, that such normal forms always exist, uses the reduction properties related to harmony. In practice, we no not always aim at normal form proofs, as these may be long and impractical. But trying to build derivations with introduction rules from conclusions and elimination rules from hypotheses is a good strategy overall.

Before showing some additional example proofs, we introduce useful notation. First of all we write

$$J_1, \ldots, J_2 \vdash J$$

for the judgment that J follows from the hypotheses $J_!, \ldots, J_n$. We call such an expression a *sequent*. More informally, we write

$$\varphi_1, \ldots, \varphi_n \vdash \varphi$$

by stripping the schematic variables, or even complex formulas, from judgments. This allows us to write, for instance,

$$\varphi \to \psi, \psi \to \chi \vdash \varphi \to \chi$$

for the judgment in Example 2.5, and more compactly

$$\Gamma \vdash \varphi,$$

if Γ is a set of hypotheses. Hypothetical judgments and sequents can of course be presented as inference rules—if the hypotheses are true, then so is the conclusion. We can see

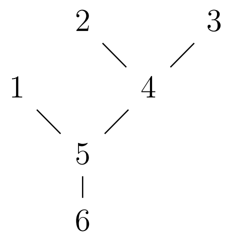
$$\varphi_1, \dots, \varphi_n \vdash \varphi$$

as a linear representation and the inference rule

$$\frac{\varphi_1 \qquad \dots \qquad \varphi_n}{\varphi}$$

with n premises as a tree representation of the same hypothetical judgment. We call such inference rules *derivable* and will return to them in Section 2.5.

Secondly, and similar to this, we can present entire proof trees in linear form. Algorithmically, this can be obtained by computing their postorder traversal. The tree



for instance, is thus translated into the list $[1, 2, 3, 4, 5, 6]$.

However, the local structure of trees with respect to inference rules may be lost in translation. To preserve this information, which helps us to justify the inferences that generated the list

$$[\psi \to \chi, \varphi \to \psi, \varphi, \psi, \chi, \varphi \to \chi]$$

and managing the discharge of hypotheses, we put derivations from such premises up to (and excluding) the conclusion obtained into boxes. Once more this is best explained by example.

Example 2.6. In linear style, the proof of

$$\varphi \to \psi, \psi \to \chi \vdash \varphi \to \chi,$$

from Example 2.5 looks as follows.

1.	$\psi \to \chi$	hyp
2.	$\varphi \to \psi$	hyp
3.	φ	hyp
4.	ψ	$\to E$, 2,3
5.	χ	$\to E$, 1,4
6.	$\varphi \to \chi$	$\to I$, 3-5

The process that leads to it can be described as follows.

1. Write down the two hypotheses (1) and (2) at the top of a piece of paper and the proof goal at its bottom (yet without a line number), leaving a big gap in the middle. The task is to fill in the gap like when completing the corresponding proof tree.

2. To proceed with $(\to I)$, open a big box, write hypothesis (3) at its top and χ at its bottom. Then add the justification $\to I$ to the last line.

3. To apply $(\to E)$ to the hypotheses in line (2) and (3); write ψ in line four, and add the line number and justification. As the justification not only write down the name of the inference used, but also the lines from which its premises were taken.

4. Apply $(\to E)$ a second time, using the expressions from line (1) and (4), and proceeding exactly like in Step 3.

5. Finally, add the missing line numbers and justifications.

□

The hypotheses could have been introduced in any order and at any time they are needed in the proof—the structural rules of natural deduction allow this. However it is often convenient and probably good style in general to introduce all "global" hypotheses at the beginning of the proof outside of all boxes.

Lemma 2.7. $\vdash ((\varphi \to \psi) \wedge (\psi \to \chi)) \to (\varphi \to \chi)$.

Proof.

1.	$(\varphi \to \psi) \wedge (\psi \to \chi)$	hyp
2.	$\varphi \to \psi$	$\wedge E_l$, 1
3.	$\psi \to \chi$	$\wedge E_r$, 1
4.	φ	hyp
5.	ψ	$\to E$, 2,4
6.	χ	$\to E$, 3,5
7.	$\varphi \to \chi$	$\to I$, 4-6
8.	$((\varphi \to \psi) \wedge (\psi \to \chi)) \to (\varphi \to \chi)$	$\to I$, 1-7

□

The lines (2)-(7) in this proof correspond precisely to the lines (1)-(6) in the proof in Example 2.6. More generally, we can translate easily between sequents

$$\varphi_1, \ldots, \varphi_n \vdash \varphi \qquad \text{and} \qquad \vdash \varphi_1 \to (\varphi_2 \to \ldots (\varphi_n \to \varphi) \ldots).$$

On the one hand, $\varphi_1, \ldots \varphi_n, \varphi_{n+1} \vdash \varphi$ implies $\vdash \varphi_1, \ldots \varphi_n \vdash \varphi_{n+1} \to \varphi$ by $(\to I)$. On the other hand, $\vdash \varphi_1, \ldots \varphi_n \vdash \varphi_{n+1} \to \varphi$ implies $\vdash \varphi_1, \ldots \varphi_n, \varphi_{n+1} \vdash \varphi$ by $(\to E)$. The above claim follows from iterating this translation or, more formally, by induction on n both ways.

In practice, the first format leads to more compact proofs. Sequents of the form

$$\vdash \varphi$$

are called *theorems*. In addition, we write

$$\varphi \dashv\vdash \psi$$

whenever $\varphi \vdash \psi$ and $\psi \vdash \varphi$. In this case, φ and ψ are called *equivalent*. According to our previous remarks, $\varphi \dashv\vdash \psi$ if and only if $\vdash \varphi \to \psi$ and $\vdash \psi \to \varphi$. This can be abbreviated by defining a new propositional connective

$$\varphi \leftrightarrow \psi = (\varphi \to \psi) \wedge (\psi \to \varphi),$$

called *biconditional*. By definition, it has the following properties.

Lemma 2.8.

1. $\varphi \to \psi, \psi \to \varphi \vdash \varphi \leftrightarrow \psi$.
2. $\varphi \leftrightarrow \psi \vdash \varphi \to \psi$.
3. $\varphi \leftrightarrow \psi \vdash \psi \to \varphi$.

For the reminder of this section we consider additional example proofs that explain various features of the natural deduction method.

Lemma 2.9.

1. $\vdash \varphi \to \varphi$.
2. $\vdash \varphi \to (\psi \to \varphi)$.

Proof.

1. The proof uses the hypothesis also as a conclusion.

$$\begin{array}{ll} 1. & \boxed{\varphi \qquad\qquad \text{hyp}} \\ 2. & \varphi \to \varphi \qquad \to I,\, 1 \end{array}$$

2. The proof uses a hypothesis multiple times.

1.	φ	hyp
2.	ψ	hyp
3.	φ	copied from 1
4.	$\psi \to \varphi$	$\to I$, 2-3
5.	$\varphi \to (\psi \to \varphi)$	$\to I$, 1-4

□

Lemma 2.10.

1. $\varphi \to \psi \vdash (\varphi \wedge \chi) \to (\psi \wedge \chi)$.

2. $\varphi \to \psi \vdash (\varphi \vee \chi) \to (\psi \vee \chi)$.

Proof.

1. The proof demonstrates conjunction introduction and elimination.

1.	$\varphi \to \psi$	hyp
2.	$\varphi \wedge \chi$	hyp
3.	φ	$\wedge E_l$, 2
4.	χ	$\wedge E_r$, 2
5.	ψ	$\to E$, 1,3
6.	$\psi \wedge \chi$	$\wedge I$ 5,4
7.	$(\varphi \wedge \chi) \to (\psi \wedge \chi)$	$\to I$, 2-6

2. The proof demonstrates disjunction introduction and elimination.

1.	$\varphi \to \psi$	hyp
2.	$\varphi \vee \chi$	hyp
3.	φ	hyp
4.	ψ	$\to E$, 1,3
5.	$\psi \vee \chi$	$\vee I_r$, 4
6.	χ	hyp
7.	$\psi \vee \chi$	$\vee I_l$, 7
8.	$\psi \vee \chi$	$\vee E$, 2,3-5,6-7
9.	$(\varphi \vee \chi) \to (\psi \vee \chi)$	$\to I$, 2-8

The rule $(\vee E)$ has three premises. Accordingly there are three references to premises in the last line of the proof.

$\square$

Lemma 2.11.

1. $\varphi \vdash \neg\neg\varphi$.

2. $\varphi \to \psi \vdash \neg\psi \to \neg\varphi$.

3. $\varphi \to \psi, \neg\psi \vdash \neg\varphi$.

Proof.

1.

$$\begin{array}{l}
\begin{array}{lll} 1. & \varphi & \text{hyp} \end{array} \\
\begin{array}{|lll|}\hline 2. & \neg\varphi & \text{hyp} \\ 3. & \bot & \neg E,\ 1,2 \\ \hline \end{array} \\
\begin{array}{lll} 4. & \neg\neg\varphi & \neg I,\ 2\text{-}3 \end{array}
\end{array}$$

2.

$$\begin{array}{l}
\begin{array}{lll} 1. & \varphi \to \psi & \text{hyp} \end{array} \\
\begin{array}{|l|}\hline
\begin{array}{lll} 2. & \neg\psi & \text{hyp} \end{array} \\
\begin{array}{|lll|}\hline 3. & \varphi & \text{hyp} \\ 4. & \psi & \to E,\ 1,3 \\ 5. & \bot & \neg E,\ 2,4 \\ \hline \end{array} \\
\begin{array}{lll} 6. & \neg\varphi & \neg I,\ 3\text{-}5 \end{array} \\ \hline
\end{array} \\
\begin{array}{lll} 7. & \neg\psi \to \neg\varphi & \to I,\ 2\text{-}6 \end{array}
\end{array}$$

3.

$$\begin{array}{l}
\begin{array}{lll} 1. & \varphi \to \psi & \text{hyp} \\ 2. & \neg\psi & \text{hyp} \end{array} \\
\begin{array}{|lll|}\hline 3. & \varphi & \text{hyp} \\ 4. & \psi & \to E,\ 1,3 \\ 5. & \bot & \neg E,\ 4,2 \\ \hline \end{array} \\
\begin{array}{lll} 6. & \neg\varphi & \neg I,\ 3\text{-}5 \end{array}
\end{array}$$

$\square$

2.5 Derived Rules

Mathematicians rarely perform proofs from basic principles. Natural deduction supports this practice. Firstly, one can freely use theorems, which have already been proved, as hypotheses. Secondly, natural deduction is open in the sense that additional inference rules can be derived and used as shortcuts in proofs. By the discussion in Section 2.4 we can derive an inference rule

$$\frac{\varphi_1 \qquad \dots \qquad \varphi_n}{\psi}$$

from every sequent or hypothetical judgment $\varphi_1, \dots, \varphi_n \vdash \psi$. Some of the facts we derived in Section 2.4 are important enough to be presented as such rules. We obtain, for instance, the following rules for implication from Section 2.3.3, Example 2.5 and Lemma 2.10.

$$\frac{}{\varphi \to \varphi} \qquad \frac{\varphi \to \psi \qquad \psi \to \chi}{\varphi \to \chi} \qquad \frac{\varphi \to \psi}{(\varphi \wedge \chi) \to (\psi \wedge \chi)} \qquad \frac{\varphi \to \psi}{(\varphi \vee \chi) \to (\psi \vee \chi)}$$

Instead of naming these rules, we may refer directly to the numbers of the Lemma where they have been proved.

The following derived rules, translated from Lemma 2.8, introduce and eliminate biconditionals, though they are not introduction or elimination rules in a strict sense.

$$\frac{\varphi \to \psi \qquad \psi \to \varphi}{\varphi \leftrightarrow \psi} \leftrightarrow 1 \qquad \frac{\varphi \leftrightarrow \psi}{\varphi \to \psi} \leftrightarrow 2 \qquad \frac{\varphi \leftrightarrow \psi}{\psi \to \varphi} \leftrightarrow 3$$

The inference rules corresponding to Lemma 2.11 are important enough to have names.

$$\frac{\varphi}{\neg\neg\varphi} \neg\neg I \qquad \frac{\varphi \to \psi}{\neg\psi \to \neg\varphi} \text{tp} \qquad \frac{p \to q \qquad \neg q}{\neg p} \text{mt}$$

The first one is called *double negation introduction*. A corresponding elimination rule is only available in classical propositional logic. The second on is sometimes called *transposition*; the last one is known as *modus tollens*.

Finally, we provide an example derivation that uses derived rules.

Example 2.12. The proof of modus tollens $\varphi \to \psi, \neg\psi \vdash \neg\varphi$ is simpler with transposition.

1.	$\varphi \to \psi$	hyp
2.	$\neg\psi$	hyp
3.	$\neg\psi \to \neg\varphi$	tp, 1
4.	$\neg\varphi$	$\to E$ 3,2

□

2.6 The Classical Rules

As already mentioned, the inference rules of natural deduction presented so far define constructive propositional logic. These show nice symmetries between introduction and elimination rules, but some widely used inference rules from mathematics are missing—the so-called *classical rules* of natural deduction.

$$\frac{}{\varphi \vee \neg\varphi}\,\text{lem} \qquad\qquad \frac{\begin{array}{c}[\neg\varphi]\\ \vdots\\ \bot\end{array}}{\varphi}\,\text{pbc} \qquad\qquad \frac{\neg\neg\varphi}{\varphi}\,\neg\neg E$$

The rule (lem) is known as *law of excluded middle* or *tertium non datur*. It states that φ, and therefore every proposition, is either true or false. It is neither an introduction nor an elimination rule, hence the symmetry of constructive natural deduction is broken. In fact, it may not even be considered an inference rules, but an *axiom*, which is a hypothesis that may always be assumed because it is not hypothetical in nature. In addition, (lem) has no computational interpretation, because it is not reasonable to assume that we always have a proof of either φ or its negation φ. In fact there are many famous open conjectures in mathematics for which we have neither.

(lem) is not derivable from the other inference rules of natural deduction; it is independent of these rules. It is therefore not valid in constructive logic, but it may still hold in particular cases. Every integer n, for instance, satisfies $(n \leq 0) \vee \neg(n \leq 0)$.

The rule (pbc) formalises the *principle of proof by contradiction*: for deriving a judgment, it suffices to derive inconsistency from its negation. This rule and proof strategy is also known as *reduction to the absurd* or *reductio ad absurdum*.

Finally, $(\neg\neg E)$ is symmetric to $(\neg\neg I)$, which we have derived in Lemma 2.11(1).

Theorem 2.13. *The rules (lem), (pbc) and $(\neg\neg E)$ are equivalent.*

Proof. Exercise. □

Natural deduction for *classical propositional logic* thus consists of the rules of intuitionistic national deduction plus one of the classical rules added. The remaining two rules are then derivable. As a rule of thumb, classical proofs tend to be more intricate than constructive ones. It is therefore valuable to have all three rules available.

Lemma 2.14. $\neg\varphi \to \neg\psi \vdash \psi \to \varphi$.

Proof.

1.	$\neg\varphi \to \neg\psi$	hyp
2.	ψ	hyp
3.	$\neg\varphi$	hyp
4.	$\neg\psi$	$\to E$, 1,3
5.	$\bot$	$\neg E$, 2,4
6.	φ	pbc, 3-5
7.	$\psi \to \varphi$	$\to I$, 2-6

□

Finally, we discuss a classical example why the law of excluded middle can be problematic.

Lemma 2.15. *There are irrational numbers m and n such that m^n is rational.*

Proof. $\sqrt{2}$ is irrational. Moreover, by he law of excluded middle, $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational. We proceed by case analysis ($\vee E$).

- If $\sqrt{2}^{\sqrt{2}}$ is rational, then the claim holds with $m = n = \sqrt{2}$.
- If $\sqrt{2}^{\sqrt{2}}$ is irrational, then the claim holds because $m^n = 2$ for $m = \sqrt{2}^{\sqrt{2}}$ and $n = \sqrt{2}$.

Thus the claim holds in general. □

The problem with this classical proof is that it does not supply any witnesses, that is, concrete irrational numbers m and n, for which m^n is rational. Is $m = n = \sqrt{2}$ the right choice? Or is it $m = \sqrt{2}^{\sqrt{2}}$ and $n = \sqrt{2}$? We cannot say! Hence the proof rather seems to pull a rabbit out of a hat...

Some mathematicians and many theoretical computer scientists would therefore object to this proof. In particularly, those acquainted with type theory, where proofs have constructive content and are strongly related to properties of programs, believe that all mathematical and computational objects should be obtained from well-defined constructions. Many mathematical textbooks and research articles, however, use classical reasoning quite extensively. In fact, we will see many such proofs in Section 2.9. Instead of insisting on constructions, they are content with definitions and proofs so long as they remain consistent. We will see in Section 2.9 that classical natural deduction preserves consistency.

Another feature of classical propositional logic is that some logical connectives becone dependent.

Lemma 2.16.

1. $\varphi \vee \psi \dashv\vdash \neg(\neg\varphi \wedge \neg\psi)$.

2. $\varphi \to \psi \dashv\vdash \neg\varphi \vee \psi$.

Proof. Exercise. □

This means that one can start with as small number of connectives, $\neg$ and $\wedge$ or $\bot$ and $\to$, say, and define the remaining connectives in terms of these. In particular, in classical propositional logic, the inference rules for the connectives defined become derivable from those one has chosen to start with. Proving this fact is an interesting exercise. An alternative combination of connectives, namely $\neg$, $\wedge$ and $\vee$, is important for automated proof search in Chapter 4.

2.7 Helpful Properties

This section lists a number of useful properties of formulas of classical propositional logic. Deriving them is an excellent exercise; all of them can be applied freely in derivations, as hypotheses or derived inference rules.

The first lemma present essentially the rules of natural deduction in sequent form.

Lemma 2.17.

1. $\bot \vdash \varphi$,
2. $\vdash \top$,
3. $\varphi, \psi \vdash \varphi \wedge \psi$,
4. $\varphi \wedge \psi \vdash \varphi$ *and* $\varphi \wedge \psi \vdash \psi$,
5. *if* $\Gamma, \varphi \vdash \psi$, *then* $\Gamma \vdash \varphi \to \psi$,
6. $\varphi \to \psi, \varphi \vdash \psi$,
7. $\varphi \vdash \varphi \vee \psi$ *and* $\psi \vdash \varphi \vee \psi$,
8. *if* $\Gamma, \phi \vdash \chi$ *and* $\Delta, \psi \vdash \chi$, *then* $\Gamma, \Delta, \varphi \vee \psi \vdash \chi$,
9. *if* $\Gamma, \phi \vdash \bot$, *then* $\Gamma \vdash \neg\phi$,
10. $\varphi, \neg\varphi \vdash \bot$,
11. $\vdash \varphi \vee \neg\varphi$.

The properties in the next lemma correspond to well known facts about boolean algebras.

Lemma 2.18.

1. $\varphi \wedge (\psi \wedge \chi) \dashv\vdash (\varphi \wedge \psi) \wedge \chi$ *(associativity of conjunction).*
2. $\varphi \wedge \psi \dashv\vdash \psi \wedge \varphi$ *(commutativity of conjunction).*
3. $\varphi \wedge \varphi \dashv\vdash \varphi$ *(idempotency of conjunction).*

4. $\varphi \vee (\psi \vee \chi) \dashv\vdash (\varphi \vee \psi) \vee \chi$ *(associativity of disjunction).*
5. $\varphi \vee \psi \dashv\vdash \psi \vee \varphi$ *(commutativity of disjunction).*
6. $\varphi \vee \varphi \dashv\vdash \varphi$. *(idempotency of disjunction).*
7. $\varphi \vee (\varphi \wedge \psi) \dashv\vdash \varphi$ *(absorption).*
8. $\varphi \wedge (\varphi \vee \psi) \dashv\vdash \varphi$ *(absorption).*
9. $\varphi \vee (\psi \wedge \chi) \dashv\vdash (\varphi \vee \psi) \wedge (\varphi \vee \chi)$ *(distributivity).*
10. $\varphi \wedge (\psi \vee \chi) \dashv\vdash (\varphi \wedge \psi) \vee (\varphi \wedge \chi)$ *(distributivity).*
11. $\varphi \wedge \neg\varphi \dashv\vdash \bot$. *(complementation)*
12. $\varphi \vee \neg\varphi \dashv\vdash \top$. *(complementation)*
13. $\varphi \vee \top \dashv\vdash \top$.
14. $\varphi \wedge \top \dashv\vdash \varphi$.
15. $\varphi \wedge \bot \dashv\vdash \bot$.
16. $\varphi \vee \bot \dashv\vdash \varphi$.
17. $\neg\bot = \top$.
18. $\neg\top \dashv\vdash \bot$.
19. $\neg(\varphi \vee \psi) \dashv\vdash \neg\varphi \wedge \neg\psi$ *(de Morgan's law).*
20. $\neg(\varphi \wedge \psi) \dashv\vdash \neg\varphi \vee \neg\psi$ *(de Morgan's law).*

The properties in the next lemma also have analogues in boolean algebra.

Lemma 2.19.

1. $\varphi \vee \psi \to \chi \dashv\vdash (\varphi \to \chi) \wedge (\psi \to \chi)$.
2. $\varphi \to (\psi \wedge \chi) \dashv\vdash (\varphi \to \psi) \wedge (\varphi \to \chi)$.
3. $\varphi \wedge \psi \to \chi \dashv\vdash (\varphi \to \chi) \vee (\psi \to \chi)$.
4. $\varphi \to (\psi \vee \chi) \dashv\vdash (\varphi \to \psi) \wedge (\varphi \to \chi)$.
5. $\varphi \wedge \neg\psi \to \chi \dashv\vdash \varphi \to \psi \vee \chi$.

The last two lemmas collect further properties.

Lemma 2.20.

1. $\varphi \to (\psi \to \chi) \vdash (\varphi \to \psi) \to (\varphi \to \chi)$*.*

2. $\neg\varphi \to \neg\psi \vdash \psi \to \phi$*.*

3. $\neg\varphi \vdash \varphi \to \psi$*.*

4. $\psi \vdash \varphi \to \psi$*.*

5. $\vdash ((\varphi \to \psi) \to \varphi \to \varphi$ *(Pierce's formula).*

Lemma 2.21.

1. $\neg\neg\varphi \dashv\vdash \varphi$ *(double negation).*

2. $\varphi \to \psi \dashv\vdash \neg\psi \to \neg\varphi$ *(contraposition).*

3. $(\varphi \wedge \psi) \to \chi \dashv\vdash \varphi \to (\psi \to \chi)$ *(curry/uncurry).*

There are various ways of relating propositional logic with boolean algebras and set-theoretic reasoning. The simplest one is to replace, e.g., $\dashv\vdash$ by $=$, $\neg$ by $-$, $\wedge$ by $\cap$, $\vee$ by $\cup$ and $\perp$ by $\emptyset$. The translation of $\to$ and $\top$ is less obvious in this context. A more systematic way is the construction of the so-called *Lindenbaum algebra* of propositional logic, which is a boolean algebra. This interesting mathematical exercise can be found in many textbook on mathematical logic or boolean algebra.

2.8 Syntax and Semantics

The introduction to natural deduction in the previous sections was by and large semi-formal. This section studies propositional logic as a mathematical theory, and even as data that we can program.

While, arguably, every scientist and mathematican can benefit from a an intuitive understanding of natural deduction, a more formal approach is particularly important for computer scientists. Some may be interested in applying logics as formal languages for modelling systems. Yet every computer scientist benefits from studying definitions of the language of propositional logic that are as precise as the specification of a programming language or a data type, and very similar to it. In addition, the formal association of meanings to logical formulas teaches computer scientists indirectly about how interpreters or compilers evaluate expressions in programs. Finally, we will define properties of propositional expressions similar to recursive functions over data types, and notions of proofs similar to the operational semantics that tell use how programs are executed step by step. Historically, of course, the origin of all these notions from computing was in logic. So here we can study them perhaps indirectly, but in a simple and pure form.

Viewed as a mathematical formalism, a *logic* consists roughly speaking of the following components:

- a *syntax* or *language* which tells us how expression can be formed;

- a *deductive systems* which transforms expressions (e.g. by inference rules);
- a *semantics* which interprets expressions, that is, gives them a meaning.

The syntax of a logic determines which particular properties or facts we can express. Propositional logic is not a very expressive logic. Predicate logic, which is introduced in Chapter 3, provides a much more fine-grained way of expressing facts. The temporal logic introduced in Chapter 5, by contrast, can express temporal properties of systems directly and concisely.

The deductive system determines what we can prove within a logic, which kind of statements we accept as axioms (e.g. the law of excluded middle), and which ones as hypothetical judgments (e.g. modus ponens). As proofs can be seen as syntactic objects, too, deductive systems are often considered to be part of the syntax.

The semantics determines ultimately what is true and what is false—at least for classical logics. For propositional logic it is really as simple as this; for predicate logic and temporal logics the semantics become somewhat more complicated.

Like the introduction and elimination rules of natural deduction, the semantics and deductive system of a logic are supposed to be in harmony: The deductive system should not be too strong; it should not allow us to derive false conclusions from true hypotheses. Yet it should ideally be strong enough to derive all true facts from true hypotheses.

Defining a logic requires another language to write down these definitions—the *metalanguage* in which we define the logic as an *object language*, in which we speak about the object language. In our case, the metalanguage is English, and we use very simple formulaic English to obtain a high level of precision. In principle, nothing would prevent us for using a formal metalanguage for defining a logic as an object language, for instance a programming language, or even the logic itself if it is expressive enough.

When writing in the metalanguage about an object language, it can be very tedious to distinguish these two levels. In natural language we sometimes use quotes to lift parts of the object language to the meta-level, for instance in the sentence "'Bird' is a word". We (re)introduce Greek letters such as $\varphi, \psi, \chi, \ldots$ as *metavariables* which range over arbitrary logical formulas we define. These are essentially the schematic variables we have used so far. Their use will become more apparent in the following paragraphs.

The metalanguage is also used for reasoning semi-formally about the object language, to perform mathematical proofs about it. The rules of natural deduction will appear implicitly at that level, in particular in the proofs of Section 2.9, where they often occur together with natural deduction rules at the object level. Whether or not this is a circularity is once more a philosophical question.

2.8.1 Syntax

Definition 2.22 (Alphabet)**.** The *alphabet* of propositional logic consists of the following symbols:

- *propositional variables*: $p, q, r, s, \ldots$ from a (countably infinite) set P;

- *the propositional connectives* : $\bot$, $\top$, $\neg$, $\vee$, $\wedge$, $\rightarrow$;
- *auxiliary symbols*: (,), ...

The propositional variables are the basic building blocks from which propositional formulas can be built.

We call $\bot$ and $\top$ *nullary* propositional connectives, $\neg$ is a *unary* connective and $\vee$, $\wedge$ and $\rightarrow$ are *binary* ones. More generally, nullary, unary and binary indicate the *arity* of a logical connective; the number of parameters it can take. Every connective has a fixed arity, though we do not make that explicit.

We use brackets like in arithmetics to make formulas unambiguous.

Definition 2.23 (Formulas)**.**

1. The set Φ of *formulas* of propositional logic (or *propositional formulas*) is defined by the following grammar (or syntax). For all $p \in P$,
$$\Phi ::= \bot \mid \top \mid p \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \rightarrow \Phi).$$
2. The propositional variables in P are called *atomic formulas*;
3. all other formulas are called *complex* or *composite*.

The expression $\Phi ::= \dots$ that defines the set of propositional formulas should be read as a recursive definition—or data type in the parlance of computer science. It says that Φ is the least set such that $\bot$, $\top$ and $p \in P$ are all in Φ. Moreover, for each formula in Φ, its negation is also in Φ, and for each pair of formulas in Φ, their conjunction, disjunction and implication is in Φ.

Obviously, the semi-formal formation rule

$$\frac{\varphi\ \mathit{prop} \qquad \psi\ \mathit{prop}}{\varphi \wedge \psi\ \mathit{prop}}\wedge F$$

corresponds to one clause in the grammar above and the other inductive clauses for $\neg$, $\vee$ and $\rightarrow$ give us other formation rules. The missing bit so far is the explicit use of atomic formulas beyond $\bot$ and $\top$.

The syntax for formulas allows us to construct the composite propositional formulas step by step from the atomic ones. The condition of Φ being the *least* such set prevents us from throwing additional junk into this set.

In the construction of composite formulas we have pedantically added brackets to make formulas unambiguous. In practice, this becomes a nuisance quite quickly.

We therefore introduce a *precedence* on propositional connecties to keep the number of brackets down. It is based on the following rules.

Definition 2.24 (Operator Precedences)**.** Among the connectives $\neg$, $\wedge$, $\vee$ and $\rightarrow$,

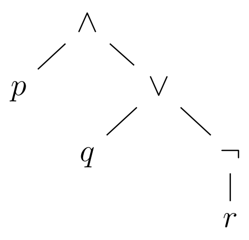
1. $\neg$ binds more strongly than the binary connectives;

2. $\wedge$ and $\vee$ have equal precedence, and they bind more strongly than $\rightarrow$.

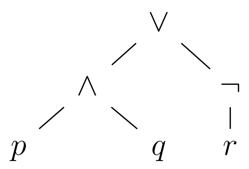
The recursive definition of the propositional formula data type ressembles that of a tree. Its root and inner nodes are labelled by propositional connectives, leaves are labelled by atomic formulas. Whether there is unary or binary branching depends on the arity of the propositional connective. In some sense we can say that a formula *is* its parse tree.

Example 2.25.

1. The parse tree of the formula $(p \wedge (q \vee (\neg r)))$ is



2. The parse tree of the formula $((p \wedge q) \vee (\neg r)))$ is



3. The string $p \wedge q \vee r$ does not have a unique parse tree; it is not clear whether $\wedge$ or $\vee$ should label its root. This is not a well-formed propositional formula!

To convert a linear textual representation of a formula, which is merely a string of symbols, into a parse tree, we benefit from the recursive definition of formulas. We can obtain most properties as recursive functions over that recursive data type. In particular, we can obtain parse trees by recursion over formulas—which is a nice programming exercise (and, more generally, the subject of parsing). The brackets and precedences used guarantee that parse trees of propositional formulas are uniquely defined.

For each string x built from the alphabet of propositional logic, it is easy to check whether $x \in \Phi$. This is the case if and only if the construction of the parse tree succeeds. The same method is used for checking whether some expression written in a programming language is an element of a given data type. This is known as type checking.

For the converse direction, the conversion of a parse tree into a string representation of a formula, one can use the inorder traversal function, one of the most basic functions for manipulating trees—up to some minor adaptations. This is another interesting programming exercise.

Here are three further examples of recursive functions over propositional formulas.

Definition 2.26 (Subformulas)**.** Let $\mathit{Sf} : \Phi \to \mathcal{P}(\Phi)$ defined by

$$\begin{aligned} \mathit{Sf}(\varphi) &= \{\varphi\}, \quad \text{if } \varphi \text{ is atomic,} \\ \mathit{Sf}(\neg\varphi) &= \{\varphi\} \cup \mathit{Sf}(\varphi), \\ \mathit{Sf}(\varphi \diamond \psi) &= \{\varphi \diamond \psi\} \cup \mathit{Sf}(\varphi) \cup \mathit{Sf}(\psi), \quad \text{if } \diamond \in \{\wedge, \vee, \to\}. \end{aligned}$$

Then, for every $\varphi \in \Phi$, $\mathit{Sf}(\varphi)$ is the set of all *subformulas* of φ.

In this definition $\mathcal{P}(\Phi)$ stands for the *power set* of Φ. In general, for every set S, $\mathcal{P}(S) = \{X \mid X \subseteq S\}$ is formed by the set of all subsets of S.

Intuitively, the subformulas of a formula are precisely the subtrees of he parse tree of the formula, including the full parse tree.

Example 2.27.

$$\begin{aligned} \mathit{Sf}((p \wedge (q \vee (\neg r)))) &= \{(p \wedge (q \vee (\neg r))))\} \cup \mathit{Sf}(p) \cup \mathit{Sf}((q \vee (\neg r))) \\ &= \{(p \wedge (q \vee (\neg r))))\} \cup \{p\} \cup \{(q \vee (\neg r))\} \cup \mathit{Sf}(q) \cup \mathit{Sf}(\neg r)) \\ &= \{(p \wedge (q \vee (\neg r)))), p, (q \vee (\neg r))\} \cup \{q\} \cup \{(\neg r)\} \cup \mathit{Sf}(r) \\ &= \{(p \wedge (q \vee (\neg r))), p, (q \vee (\neg r)), q, (\neg r)\} \cup \{r\} \\ &= \{(p \wedge (q \vee (\neg r))), p, (q \vee (\neg r)), q, (\neg r), r\} \end{aligned}$$

□

Definition 2.28 (Height)**.** The *height* of a propositional formula is a function $h : \Phi \to \mathbb{N}$ defined by

$$\begin{aligned} h(\varphi) &= 1, \quad \text{if } \varphi \text{ is atomic,} \\ h(\neg\varphi) &= 1 + h(\varphi), \\ h(\varphi \diamond \psi) &= 1 + \max(h(\varphi), h(\psi)), \quad \text{if } \diamond \in \{\wedge, \vee, \to\}. \end{aligned}$$

Intuitively, the height of a formula is the length of the longest branch in its parse tree, in the sense that it is equal to the number of nodes on that branch.

Example 2.29. $h((p \wedge (q \vee (\neg r)))) = 4.$ □

Definition 2.30.

1. The function $V : \Phi \to \mathcal{P}(P)$ defined by

$$\begin{aligned} V(\varphi) &= \{\varphi\}, \quad \text{if } \varphi \text{ is atomic,} \\ V(\neg\varphi) &= V(\varphi), \\ V(\varphi \diamond \psi) &= V(\varphi) \cup V(\psi), \quad \text{if } \diamond \in \{\wedge, \vee, \to\}, \end{aligned}$$

   computes the set of atoms that occur in a given propositional formula.

2. A propositional variable p *occurs* in a formula φ if $p \in V(\varphi)$.

Another advantage of the recursive definition of formulas is that properties can be verified by (structural) induction.

Lemma 2.31. *$V(\varphi) = Sf(\varphi) \cap P$ holds or all $\varphi \in \Phi$,*

Proof. By structural induction over φ. In the base case, if $\varphi \in P$, then $\{\varphi\} \cap P = \{\varphi\}$ and therefore $V(\varphi) = \{\varphi\} = \{\varphi\} \cap P = Sf(\varphi) \cap P$.

In the induction step we distinguish between unary and binary connectives. Firstly, if $\varphi = \neg\psi$, then $V(\psi) = Sf(\psi) \cap P$ holds by the induction hypothesis, and

$$\begin{aligned} V(\neg\psi) &= V(\psi) \\ &= Sf(\psi) \cap P \\ &= (\{(\neg\psi)\} \cap P) \cup (Sf(\psi) \cap P) \\ &= (\{(\neg\psi)\} \cup Sf(\psi)) \cap P \\ &= Sf((\neg\psi)) \cap P, \end{aligned}$$

because $\{(\neg\psi)\} \cap P = \emptyset$.

Secondly, if $\varphi = \psi \diamond \chi$, then $V(\psi) = Sf(\psi) \cap P$ and $V(\chi) = Sf(\chi) \cap P$ by the induction hypothesis. Hence

$$\begin{aligned} V(\psi \diamond \chi) &= V(\psi) \cup V(\chi) \\ &= (Sf(\psi) \cap P) \cup (Sf(\chi) \cap P) \\ &= (\{\psi \diamond \chi\} \cap P) \cup (Sf(\psi) \cap P) \cup (Sf(\chi) \cap P) \\ &= (\{\psi \diamond \chi\} \cup Sf(\psi) \cup Sf(\chi)) \cap P \\ &= Sf(\psi \diamond \chi) \cap P. \end{aligned}$$

□

=== END OF WEEK THREE ====================================

2.8.2 Semantics

Under the classical interpretation, propositional logic deals with judgments of the form φ *true*. The meaning of a propositional formula is thus its *truth value*: true or false. Moreover, the meaning of a compound formula depends only on the meaning of its top propositional connective and that of its subformulas. It can thus be determined recursively from the meaning of the subformulas.

This recursion is captured by the truth tables for the propositional connectives. Here and henceforth, we write 1 for the truth value "true" and 0 for the truth value "false". We also call the set $\mathbb{B} = \{0, 1\}$ the *booleans*.

φ	$\neg\varphi$
1	0
0	1

φ	ψ	$\varphi \wedge \psi$
1	1	1
1	0	0
0	1	0
0	0	0

φ	ψ	$\varphi \vee \psi$
1	1	1
1	0	1
0	1	1
0	0	0

φ	ψ	$\varphi \to \psi$
1	1	1
1	0	0
0	1	1
0	0	1

Definition 2.32 (Assignment)**.** An *assignment* is a function $v : P \to \mathbb{B}$ from the set P of propositional variables into the truth values in $\mathbb{B}$.

By virtue of the recursive definition of propositional formulas, assignments can be extended to propositional formulas by recursion.

Definition 2.33 (Valuation)**.** Let $v : P \to \mathbb{B}$ be a assignment. A *valuation* is a function $[\![-]\!]_v : \Phi \to \mathbb{B}$ defined by

$$\begin{aligned}
[\![\top]\!]_v &= 1, \\
[\![\bot]\!]_v &= 0, \\
[\![p]\!]_v &= v(p), \quad \text{if } p \in P, \\
[\![\neg\varphi]\!]_v &= 1 - [\![\varphi]\!]_v, \\
[\![\varphi \wedge \psi]\!]_v &= \min([\![\varphi]\!]_v, [\![\psi]\!]_v), \\
[\![\varphi \vee \psi]\!]_v &= \max([\![\varphi]\!]_v, [\![\psi]\!]_v), \\
[\![\varphi \to \psi]\!]_v &= \max(1 - [\![\varphi]\!]_v, [\![\psi]\!]_v).
\end{aligned}$$

Valuations are also called *interpretations*. They are completely determined by the underlying assignments.

Lemma 2.34. *Let $\varphi \in \Phi$ and v and w be assignments such that $v(p) = w(p)$, for all $p \in V(\varphi)$. Then $[\![\varphi]\!]_v = [\![\varphi]\!]_w$.*

Proof. By structural induction. □

In other words, the truth value of a propositional formula is determined completely by the assignment of truth values to its variables.

It is a valuable exercise to compare valuations and truth tables. Each row in a truth table for a certain connective corresponds to the recursive case in the definition of the connective for a fixed assignment. Suppose, for instance, that $[\![\varphi]\!]_v = 1$ for a given assigment v of truth values to the variables in φ. Then $[\![\neg\varphi]\!]_v = 0$, which corresponds to the first row in the truth table for negation.

We henceforth write $[\![-]\!]$ when the particular valuation does not matter.

Lemma 2.35.

1. $[\![\neg\varphi]\!] = 1$ *if and only if* $[\![\varphi]\!]$ *is not* 1.

2. $[\![\varphi \wedge \psi]\!] = 1$ *if and only if* $[\![\varphi]\!] = 1$ *and* $[\![\psi]\!] = 1$.

3. $[\![\varphi \vee \psi]\!] = 1$ *if and only if* $[\![\varphi]\!] = 1$ *or* $[\![\psi]\!] = 1$.

4. $[\![\varphi \to \psi]\!] = 0$ *if and only if,* $[\![\varphi]\!] = 1$ *and* $[\![\psi]\!] = 0$.

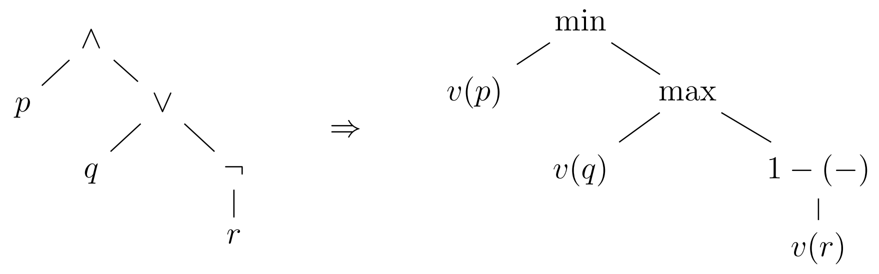
Apart from $\to$, the metalevel statements at the right-hand side of the "if and only if" clauses correspond directly to the meaning of the object-level connectives. Case (1), for instance, says that a formula $\neg\varphi$ is true if and only if φ is not true; case (2) says that $\varphi \wedge \psi$ is true if and only if φ is true and ψ is true. In this sense, the semantics of classical propositional logic is almost trivial.

For a given assignment, the recursive definition of $[\![-]\!]$ allows us to compute the truth value of a formula recursively, like we evaluate the value of an expression (e.g. an arithmetic expression) in a program.

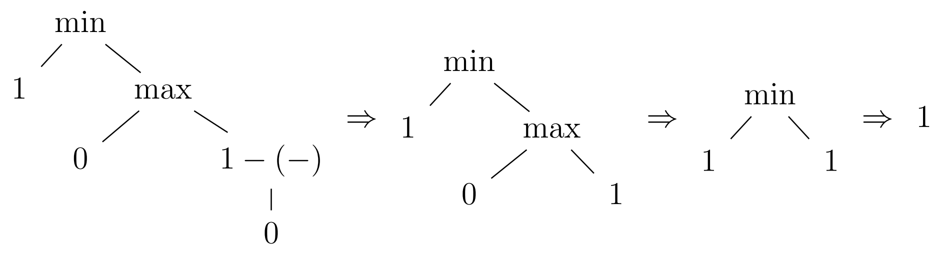
Example 2.36. Let $v(p) = 1$ and $v(q) = v(r) = 0$ and consider the formula $p \wedge (q \vee \neg r)$. Then

$$\begin{aligned}
[\![p \wedge (q \vee \neg r)]\!] &= \min([\![p]\!], [\![q \vee \neg r]\!]) \\
&= \min(v(p), \max([\![q]\!], [\![\neg r]\!])) \\
&= \min(1, \max(v(q), 1 - [\![r]\!])) \\
&= \max(0, 1 - v(r))) \\
&= 1 - 0 \\
&= 1.
\end{aligned}$$

From a computer science point of view, we simply evaluate the propositional formula in the booleans, we compute its boolean value with a recursive program. This means that we wrap the recursive definition over the parse tree of the formulas



and then propagate values in bottom-up fashion.



This bottom-up propagation of values corresponds precisely to how we would evaluate a row in the truth table of $p \wedge (q \vee \neg r)$. □

At the end of this section we introduce some important semantic concepts and notation.

Definition 2.37 (Tautologies/Entailment/Equivalence)**.**

1. A formula $\varphi \in \Phi$ is a *tautology* if $[\![\varphi]\!]_v = 1$ for all assignments v. We write $\models \varphi$ if φ is a tautology.
2. A set $\Gamma \subseteq \Phi$ of propositional formulas *(semantically) entails* a formula $\varphi \in \Phi$, written $\Gamma \models \varphi$, if, for every assignment v, if $[\![\psi]\!]_v = 1$ holds for all $\psi \in \Gamma$, then $[\![\varphi]\!]_v = 1$.
3. Two formulas $\varphi, \psi \in \Phi$ are *logically equivalent*, written $\varphi \equiv \psi$, if $[\![\varphi]\!]_v = [\![\psi]\!]_v$ holds for every v.

Tautologies are also called *valid* formulas. The relation $\models$ of semantic entailment is also called the *semantic consequence* relation. Intuitively, $\Gamma \models \phi$ holds if ϕ is true whenever all formulas in Γ are. It follows that φ is a tautology if and only if $\emptyset \models \varphi$.

We usually write $\varphi_1, \ldots \varphi_n \models \varphi$ instead of $\{\varphi_1, \ldots \varphi_n\} \models \varphi$ and $\Gamma, \varphi \models \psi$ instead of $\Gamma \cup \{\varphi\} \models \psi$. Intuitively, $\varphi \models \psi$ if ψ is at least as true as ϕ.

Definition 2.38 (Satisfiability)**.**

1. A formula $\varphi \in \Phi$ is *satisfiable* if $[\![\varphi]\!]_v = 1$ for some assignment v.
2. φ is *unsatisfiable* (a *contradiction*) if is is not satisfiable.
3. A set $\Gamma \subseteq \Phi$ is *satisfiable* if every $\varphi \in \Gamma$ is.
4. A set $\Gamma \subseteq \Phi$ is *unsatisfiable* if some $\varphi \in \Gamma$ is.

Lemma 2.39. *A formula φ is valid if and only if $\neg\varphi$ is unsatisfiable.*

The so-called SAT problem asks to decide whether or not there is an interpretation v that satisfies a given formula φ, that is, $[\![\varphi]\!]_v = 1$. A solution to this problem is an algorithm that takes propositional formulas as an input and returns "yes" if there is an assignment v for which $[\![\varphi]\!]_v = 1$. Otherwise, if $[\![\varphi]\!]_v = 0$ for all assignments v, it returns "no".

The SAT problem was the first known NP-complete problem, which means that tractable solutions, those which would require polynomial time in the size of the input formula, do not exist (unless P=NP).

We know how to solve the SAT problem: we can compute the truth table of any propositional formula and decide whether or not one row evaluates to 1. But this brute force methods requires time exponential in the size of the input formula (if we can guess the right row, we can check its value in polynomial time). By NP-completness, we even cannot expect to do much better than that. In practice, however, so-called SAT-solvers can tackle propositional formulas with more than $10k$ variables successfully. We will study such algorithms in Chapter 4. A deeper study of SAT problems in particular, and NP-complete problems, in general, is the aim of *computational complexity* theory.

We end this section with some semantic entailments.

Lemma 2.40.

1. *$\bot \models \varphi$,*
2. *$\models \top$,*
3. *$\varphi, \psi \models \varphi \wedge \psi$,*
4. *$\varphi \wedge \psi \models \varphi$ and $\varphi \wedge \psi \vdash \psi$,*
5. *if $\Gamma, \varphi \models \psi$, then $\Gamma \vdash \varphi \to \psi$,*
6. *$\varphi \to \psi, \varphi \models \psi$,*
7. *$\varphi \models \varphi \vee \psi$ and $\psi \models \varphi \vee \psi$,*
8. *if $\Gamma, \phi \models \chi$ and $\Delta, \psi \models \chi$, then $\Gamma, \Delta, \varphi \vee \psi \models \chi$,*
9. *if $\Gamma, \phi \models \bot$, then $\Gamma \models \neg\phi$,*
10. *$\varphi, \neg\varphi \models \bot$,*
11. *$\models \varphi \vee \neg\varphi$.*

Proof. We consider a few cases. (1) holds because φ is at least as true as $\bot$; (2) because $\top$ is a tautology. (3) and (4) hold because $\varphi \wedge \psi$ is true if and only if φ and ψ both are, by Lemma 2.35(2). For (5), suppose that ψ is true whenever Γ is true and φ is true. This means that, assuming Γ is true, ψ is true whenever φ is. Hence if Γ is true, then so is $\varphi \to \psi$. □

2.8.3 Deductive System

With the syntax of propositional logic defined, we can use the natural deduction method from Section 2.6, to reason within this logic. It is therefore important that the rules feature schematic variables or metavariables, so that we can substitute arbitrary formulas for them. Modus ponens, for instance, is of the form

$$\frac{\varphi \to \psi \qquad \varphi}{\psi}$$

The set of proof trees can be defined by recursion. For the sake of simplicity we assume the restricted syntax

$$\Phi ::= \bot \mid p \mid (\Phi \wedge \Phi) \mid (\Phi \to \Phi),$$

where $p \in P$, as in classical logic the other propositional connectives are definable. Note that this set is not even minimal.

Definition 2.41 (Proof Tree)**.** The set T of *proof trees* of natural deduction is the smallest set such that

1. For all $p \in P$, the one-element tree t labelled with p is in T.
2. For all $\varphi \in \Phi$, the one-element tree t labelled with $\varphi \vee \neg\varphi$ is in T.
3. If $t_1, t_2 \in T$ such that t_1 has root label φ and t_2 has root label ψ, then $t \in T$ with root label $\varphi \wedge \psi$ and immediate subtrees t_1 and t_2.
4. If $t' \in T$ with root label $\varphi \wedge \psi$, then $t \in T$ with root label φ and immediate subtree t'.
5. If $t' \in T$ with root label $\varphi \wedge \psi$, then $t \in T$ with root label ψ and immediate subtree t'.
6. If $t' \in T$ with root label ψ and a leaf labelled by φ (there may be other leaves), then $t \in T$ with root label $\varphi \to \psi$ and its immediate subtree is t' with leaf label φ replaced by $[\varphi]$.
7. If $t_1, t_2 \in T$ such that t_1 has root label $\varphi \to \psi$ and t_2 has root label ψ, then $t \in T$ with root label ψ and immediate subtrees t_1 and t_2.
8. If $t' \in T$ with root labelled by $\bot$, then, for any $\varphi \in \Phi$, $t \in T$ with root label φ and immediate subtree t'.

Formation rules for proof trees with the other inference rules of natural deduction can be defined in an analogous way.

Using the definition of proof tree allows us to (re)define our notation $\vdash$ of sequents as follows. For $\Gamma \subseteq \Phi$ and $\varphi \in \Phi$ we write

$$\Gamma \vdash \varphi$$

if there is a proof tree in T with root labelled by φ and all live labels of leaves in Γ. This is consistent with our previous use, which does not show local discharged hypotheses in hypothetical judgments.

Attempting a direct formal definition of linear proofs is more involved. Obviously, a linear proofs for $\Gamma \vdash \phi$ is a sequence or list $\varphi_1, \ldots, \varphi_n$ of propositional formulas such that $\varphi_n = \varphi$, and for each $k \leq n$ every ϕ_k is either a live hypothesis from Γ, a killed hypothesis (which need not be from Γ) or a formula obtained from one formula φ_i or two formulas φ_i and φ_j with $i, j < k$ by using an inference rule. But of course, while we may not use all hypotheses in Γ, we must ensure that all killed hypotheses are actually used in inference rules. Capturing this is somewhat tedious. Instead of writing $[\varphi]$ for the killed hypothesis in $(\to I)$, for instance, or putting a box around the hypothetical judgment associated with it, we could associate a label with that hypothesis and the inference rule, that is, write

$$\dfrac{\dfrac{}{\varphi}\, n \\ \vdots \\ \psi}{\varphi \to \psi} \to I_n$$

The label n thus allows use to associate killed hypotheses with inferences. The rule $(\vee E)$, would have to be associated with two labels, that is, $(\vee E_{mn})$ to indicate the two hypothetical judgments required. Managing proofs in this way is left as a programming exercise.

The following fact is an immediate consequence of the finiteness of proof trees.

Lemma 2.42. *If $\Gamma \vdash \varphi$, then $\Gamma_0 \vdash \varphi$ for some finite $\Gamma_0 \subseteq \Gamma$.*

2.9 Soundness, Completeness, Compactness

Without a clear connection between syntax and semantics, natural deduction is just a meaningless game. In the classical context, the hypothetical judgments set out in the rules of natural deduction guarantee that we derive conclusions we trust to be true from hypotheses we assume to be true. Using our notation for derivability $\Gamma \vdash \varphi$ and semantic entailment $\Gamma \models \varphi$, we should expect that

$$\Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi,$$

where $\Rightarrow$ stands for meta-level implication: If $\Gamma \vdash \varphi$ holds, then so does $\Gamma \models \varphi$. Similarly, we write $\Leftrightarrow$ for meta-level "if and only if". We call natural deduction, and more generally, every logic with a formal semantics and a deductive system, *sound* if it satisfies this property. Soundness is a crucial, but also a very weak requirement for a logic. A logic without any deductive system is certainly sound. It deduces nothing, but also does not do any harm. Instead one might ask whether a deductive system is strong enough to derive any fact that is semantically entailed by a set of hypotheses.

$$\Gamma \models \varphi \Rightarrow \Gamma \vdash \varphi.$$

This property is called *completeness* of the deductive system with respect to the semantics of a logic. Completeness cannot always been achieved—Gödel's famous incompleteness theorems give examples of logics where it fails—and is perhaps not always practically relevant. Some proofs in a complete deductive systems might be so long that we can never find and write them in practice. But completeness is certainly a natural property to ask for.

Intuitively speaking, soundness and completeness address the following questions:

- Are all conclusions we can derive from true premises true (soundness)?
- Can we derive all true conclusions from true premises (completeness)?

This section shows that propositional logic is both sound and complete. The proofs themselves are perhaps not too impressive, but they give a glimpse of how similar proofs for other logics can be obtained.

As in the previous section, it is convenient to restrict our attention to the syntax

$$\Phi ::= \bot \mid p \mid (\Phi \wedge \Phi) \mid (\Phi \to \Phi).$$

This allows us to keep the case analyses in proofs small; but of course we can freely use all rules of natural deduction , since those for disjunction and negation are derivable from those for $\bot$, $\wedge$ and $\to$.

Theorem 2.43 (Soundness of Propositional Logic)**.** *Let* $\Gamma \subseteq \Phi$ *and* $\varphi \in \Phi$*. Then*

$$\Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi.$$

Proof. We need to show that $\Gamma \models \varphi$ holds for all derivations of φ from hypotheses in Γ. We proceed by induction over the structure of proof trees. Alternatively we can see this as an induction on the number of steps in the proof. Because of the syntax restriction, it suffices to consider the introduction and elimination rules for conjunction and implication, the elimination rule for $\bot$ and the law of excluded middle in the proof.

- In the base case, the one-element proof tree, either $\varphi \in \Gamma$, from which $\Gamma \models \varphi$ follows, or $\varphi = \psi \vee \neg\psi$, which even satisfies $\emptyset \models \varphi$.

- For the induction step, we perform a case analysis on the inference rules in the last step of the derivation.

  ($\wedge I$) Then $\varphi = \psi_1 \wedge \psi_2$, the proof has the immediate subproofs $\Gamma \vdash \psi_1$ and $\Gamma \vdash \psi_2$ (we may need less hypotheses for these subproofs, but that does not matter), and $\Gamma \models \psi_1$ and $\Gamma \models \psi_2$ hold by the induction hypothesis. Hence if $[\![\chi]\!]_v = 1$ for all $\chi \in \Gamma$, then $[\![\psi_1]\!]_v = 1$ and $[\![\psi_2]\!]_v = 1$ by the definition of semantic entailment, and $[\![\psi_1 \wedge \psi_2]\!]_v = 1$ by Lemma 2.35(2). Thus $\Gamma \models \psi_1 \wedge \psi_2$.

  ($\wedge E_l$) Then the proof has the immediate subproof $\Gamma \vdash \varphi \wedge \psi$ for some $\psi \in \Phi$ and $\Gamma \models \varphi \wedge \psi$ holds by the induction hypothesis. Hence if $[\![\chi]\!]_v = 1$ for all $\chi \in \Gamma$, then $[\![\varphi \wedge \psi]\!]_v = 1$ by the definition of semantic entailment, and $[\![\varphi]\!]_v = 1$ by Lemma 2.35(2). Thus $\Gamma \models \varphi$.

  ($\wedge E_r$) The proof is similar to the previous one.

  ($\to I$) Then $\varphi = \psi_1 \to \psi_2$, the proof has the immediate subproof $\Gamma, \psi_1 \vdash \psi_2$, and $\Gamma, \psi_1 \models \psi_2$ holds by the induction hypothesis. Hence if $[\![\chi]\!]_v = 1 = [\![\psi]\!]_v$, then $[\![\psi_2]\!]_v = 1$, by the definition of semantic entailment and $[\![\psi_! \to \psi_2]\!]_v = 1$ by Lemma 2.35(4). Thus $\Gamma \models \psi_1 \to \psi_2$.

  ($\to E$) Then the proof has the immediate subproofs $\Gamma \vdash \psi \to \varphi$ and $\Gamma \vdash \psi$, and $\Gamma \models \psi \to \varphi$ and $\Gamma \models \psi$ hold by the induction hypothesis. Hence if $[\![\chi]\!]_v = 1$ for all $\chi \in \Gamma$, then $[\![\psi \to \varphi]\!]_v = 1$ and $[\![\psi]\!]_v = 1$ by the definition of entailment, and $[\![\varphi]\!]_v = 1$ by Lemma 2.35(4). Thus $\Gamma \models \varphi$.

  ($\bot E$) Then the proof has the immediate subproof $\Gamma \vdash \bot$, and $\Gamma \models \bot$ holds by the induction hypothesis. Now $[\![\bot]\!]_v = 0$ for all assignments v. Thus, for any v, there is some $\chi \in \Gamma$ such that $[\![\chi]\!]_v = 0$. But then $\Gamma \models \varphi$ for any $\varphi \in \Phi$ by definition of semantic entailment.

□

=== END OF WEEK FOUR (EXPECTED) ====================

The completeness proof is less direct. It depends on some intermediate lemmas, and puts consistency at the heart of classical logic. We follow one of the two most popular proofs. Its particular advantage is that it can be adapted to many similar situations.

Definition 2.44. A set $\Gamma \subseteq \Phi$ is *consistent* if $\Gamma \nvdash \bot$.

Hence Γ is *inconsistent* if $\Gamma \vdash \bot$.

Lemma 2.45. *The following conditions are equivalent.*

1. *Γ is consistent.*

2. *There is no $\varphi \in \Phi$ such that $\Gamma \vdash \varphi$ and $\Gamma \vdash \neg\varphi$.*

3. *There is some $\varphi \in \Phi$ such that $\Gamma \nvdash \varphi$.*

Proof.
(1) $\Rightarrow$ (2). If $\Gamma \vdash \varphi$ and $\Gamma \vdash \neg\varphi$ holds for some $\varphi \in \Phi$, then $\Gamma \vdash \bot$ by $(\neg E)$.
(2) $\Rightarrow$ (3). If $\Gamma \vdash \psi$ holds for every $\psi \in \Phi$, then in particular, $\Gamma \vdash \varphi \wedge \neg\varphi$ and therefore $\Gamma \vdash \varphi$ and $\Gamma \vdash \neg\varphi$ by $(\wedge E_l)$ and $(\wedge E_r)$.
(3) $\Rightarrow$ (1). If $\Gamma \vdash \bot$, then $\Gamma \vdash \varphi$ for every $\varphi \in \Phi$ by $(\bot E)$. □

Intuitively, consistency and satisfiability should be the same. First we prove one direction.

Lemma 2.46. *A set $\Gamma \subseteq \Phi$ is consistent if Γ is satisfiable.*

Proof. If Γ is inconsistent, then $\Gamma \vdash \bot$ and therefore $\Gamma \models \bot$ by soundness (Theorem 2.43). But as $[\![\bot]\!]_v = \bot$ for every v, there cannot be a v such that $[\![\varphi]\!]_v = 1$ for every $\varphi \in \Gamma$. Hence Γ is unsatisfiable. □

The other direction requires some work. It leads us straight to the completeness theorem.

Lemma 2.47.

1. *If $\Gamma \cup \{\varphi\}$ is inconsistent, then $\Gamma \vdash \neg\varphi$.*

2. *If $\Gamma \cup \{\neg\varphi\}$ is inconsistent, then $\Gamma \vdash \varphi$.*

Proof.

1. If $\Gamma \cup \{\varphi\}$ is inconsistent, then $\Gamma, \varphi \vdash \bot$, whence $\Gamma \vdash \neg\varphi$ by $(\neg I)$.

2. If $\Gamma \cup \{\neg\varphi\}$ is inconsistent, then $\Gamma, \neg\varphi \vdash \bot$, whence $\Gamma \vdash \varphi$ by (pbc).

□

Definition 2.48. A consistent set $\Gamma \subseteq \Phi$ is *maximally consistent* if, for each consistent set $\Delta \subseteq \Phi$, $\Gamma \subseteq \Delta$ implies $\Gamma = \Delta$.

Intuitively, a maximal consistent set is a consistent set which becomes inconsistent when further formulas are added to it.

Lemma 2.49 (Lindenbaum's Lemma)**.** *Every consistent $\Gamma \subseteq \Phi$ is contained in a maximally consistent $\Gamma^* \subseteq \Phi$.*

Proof. Suppose we have a list $\varphi_0, \varphi_1, \varphi_2, \ldots$ of all propositions This is possible since, by virtue of the inductive construction of Φ, the set of propositional is countable. Define the chain of supersets $\Gamma \subseteq \Gamma_1 \subseteq \cdots \subseteq \Gamma^*$ of Γ by

$$\Gamma_0 = \Gamma, \qquad \Gamma_{n+1} = \begin{cases} \Gamma_n \cup \{\varphi_n\}, & \text{if this set is consistent,} \\ \Gamma_n, & \text{otherwise,} \end{cases} \qquad \Gamma^* = \bigcup_{n \in \mathbb{N}} \Gamma_n.$$

Each Γ_n is consistent by construction and we show that Γ^* is consistent as well. Suppose otherwise, that is, $\Gamma^* \vdash \bot$. By Lemma 2.42 there is a finite $\Delta \subseteq \Gamma^*$ such that $\Delta \vdash \bot$. Each element of Δ entered Γ^* at some stage of the construction, hence there is a $k \in \mathbb{N}$, at which the last element of Δ entered some Γ_k. Thus $\Delta \subseteq \Gamma_k$ and therefore $\Gamma_k \vdash \bot$. But this is impossible because Γ_k is consistent—a contradiction.

Finally we show that Γ^* is maximally consistent. Hence suppose that some $\Delta \supseteq \Gamma^*$ is consistent. We show that every element $\psi \in \Delta$ is also in Γ^*. Since $\psi \in \Phi$ it must be equal to some φ_k in our list. Hence it was considered in the construction of $\Gamma_k \subseteq \Gamma^* \subseteq \Delta$. As Δ is consistent, it must have been added to $\Gamma_{k+1} \subseteq \Gamma^*$. Hence $\psi \in \Gamma^*$ and therefore $\Gamma = \Delta$. □

Lindenbaum's lemma shows how every consistent set can be extended to some maximal consistent one. Consistent sets can have several such extensions. Maximal consisten sets have many nice properties, and they appear in other guises in many areas of mathematics—for instance as ultrafilters in boolean algebras.

The next lemma shows that maximal consistent sets are closed under deduction, that is, every consequence of a maximal consistent set is already in the set.

Lemma 2.50. *If $\Gamma \subseteq \Phi$ is maximally consistent, then*

$$\Gamma \vdash \varphi \Leftrightarrow \varphi \in \Gamma.$$

Proof. If Γ is maximally consistent and $\varphi \notin \Gamma$, then $\Gamma \cup \{\varphi\}$ is inconsistent. Hence $\Gamma \vdash \neg\varphi$ by Lemma 2.47(1) and therefore $\Gamma \nvdash \varphi$ by Lemma 2.45(2). The converse direction is trivial because every hypothesis can be derived. □

More formally, we could thus write $\Gamma = \{\varphi \mid \Gamma \vdash \varphi\}$ when Γ is maximally consistent. In some sense, maximal consistent sets contain all the information about Γ we need to know. In particular enough information for building a satisfying valuation from it.

The following lemma reflects the semantic properties of Lemma 2.35. It further confirms the claim that maximal consistent sets form a good basis for building valuations.

Lemma 2.51. *Let $\Gamma \subseteq \Phi$ be maximally consistent. Then for all $\varphi, \psi \in \Phi$,*

1. *either $\varphi \in \Gamma$ or $\neg\varphi \in \Gamma$;*
2. *$\varphi \wedge \psi \in \Gamma$ if and only if $\varphi \in \Gamma$ and $\psi \in \Gamma$,*
3. *$\varphi \vee \psi \in \Gamma$ if and only if $\varphi \in \Gamma$ or $\psi \in \Gamma$,*

4. *$\varphi \to \psi \notin \Gamma$ if and only $\varphi \in \Gamma$ and $\psi \notin \Gamma$.*

Proof.

1. By Lemma 2.45(2), at most one of φ and $\neg\varphi$ can be derivable from Γ, hence in Γ by Lemma 2.50. If $\Gamma \cup \{\varphi\}$ is consistent, then $\varphi \in \Gamma$ by maximality of Γ. Otherwise, $\Gamma \vdash \neg\varphi$ by Lemma 2.47(1) and therefore $\neg\varphi \in \Gamma$ by Lemma 2.50.

2. $\varphi \wedge \psi \in \Gamma$ if and only if $\Gamma \vdash \varphi \wedge \psi$ by Lemma 2.50. This is the case if and only if $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$ by $(\wedge I)$, $(\wedge E_l)$ and $(\wedge E_r)$. This is the case if and only if $\varphi \in \Gamma$ and $\psi \in \Gamma$, again by Lemma 2.50.

3. $\varphi \vee \psi \in \Gamma$ if and only if $\neg(\neg\varphi \wedge \psi) \in \Gamma$ by Lemma 2.16(1). This is the case if and only if $\neg\varphi \wedge \neg\psi \notin \Gamma$ by (1). This is the case if and only if $\neg\varphi \notin \Gamma$ or $\neg\varphi \notin \Gamma$ by (2). This is the case if and only if $\varphi \in \Gamma$ or $\psi \in \Gamma$, again by (1).

4. With $\varphi \to \psi \notin \Gamma \Leftrightarrow \varphi \wedge \neg\psi \in \Gamma$, which is a consequence of Lemma 2.16), the proof is similar to (3), using (1) and (2).

□

The next lemma proves the converse direction to Lemma 2.46, using essentially the information from Lemma 2.51.

Lemma 2.52. *If $\Gamma \subseteq \Phi$ is consistent, then Γ is satisfiable.*

Proof. Let Γ^* be a maximal consistent set containing Γ. For each $p \in P$ define the assignment

$$v(p) = \begin{cases} 1, & \text{if } p \in \Gamma^*, \\ 0, & \text{otherwise.} \end{cases}$$

We prove that $[\![\varphi]\!]_v = 1 \Leftrightarrow \varphi \in \Gamma^*$ by induction on φ, using the restricted syntax with $\bot$, $\wedge$ and $\to$.

- If $\varphi \in P$, the claim holds by construction of v.
- Let $\varphi = \psi_1 \wedge \psi_2$. Then $[\![\varphi]\!]_v = 1 \Leftrightarrow [\![\varphi_1]\!]_v = 1 = [\![\psi_2]\!]_v$ by Lemma 2.35(2). This is the case if and only if $\psi_1, \psi_2 \in \Gamma^*$ by the induction hypothesis. Finally, this is the case if and only if $\varphi \in \Gamma^*$ by Lemma 2.51(2).
- Let $\varphi = \psi_1 \to \psi_2$. Then $[\![\varphi]\!]_v = 0$ if and only if $[\![\psi_1]\!]_v = 1$ and $[\![\psi_2]\!]_v = 0$ by Lemma 2.35(4). By the induction hypothesis, this is the case if and only if $\varphi \in \Gamma^*$ and $\psi \notin \Gamma^*$. Finally, this is the case if and only if $\varphi \notin \Gamma^*$ by Lemma 2.51(4).

Finally, $\Gamma \subseteq \Gamma^*$ and therefore $[\![\varphi]\!]_v = 1$ for all $\varphi \in \Gamma$. □

Corollary 2.53. $\Gamma \subseteq \Phi$ is consistent if and only if Γ is satisfiable.

Proof. Immediate from Lemma 2.46 and 2.52. □

This corollary is sometimes called the *small completeness theorem*. The completeness theorem of propositional logic is now a simple consequence.

Theorem 2.54 (Completeness of Propositional Logic)**.** *Let* $\Gamma \subseteq \Phi$ *and* $\varphi \in \Phi$. *Then*

$$\Gamma \models \varphi \Rightarrow \Gamma \vdash \varphi.$$

Proof. Suppose $\Gamma \nvdash \varphi$. By Lemma 2.47 and 2.45(2), this is the case if and only if $\Gamma \cup \{\neg\varphi\}$ is consistent. This is case if and only if if $\Gamma \cup \{\neg\varphi\}$ is satisfiable by Corollary 2.53, that is, there is an assignment v such that $[\![\varphi]\!]_v = 0$ and $[\![\psi]\!]_v = 1$ for all $\psi \in \Gamma$. Thus $\Gamma \not\models \varphi$. □

Theorem 2.55. *Let* $\Gamma \subseteq \Phi$ *and* $\varphi \in \Phi$. *Then*

$$\Gamma \vdash \varphi \Leftrightarrow \Gamma \models \varphi.$$

Theorem 2.56 (Compactness)**.** *A set* $\Gamma \subseteq \Phi$ *is satisfiable if and only if every finite* $\Gamma_0 \subseteq \Gamma$ *is satisfiable.*

Proof. If Γ is satisfiable, then, of course, so is any subset of Γ. For the converse direction suppose Γ is not satisfiable. Then it is inconsistent by Corollary 2.53. Thus $\perp$ can be obtained from a finite set of hypotheses Γ_0 from Γ by Lemma 2.42. Thus already Γ_0 is inconsistent and therefore unsatisfiable, once more by Corollary 2.53. □

The compactness theorem is often applied in the following form: if a set of propositional formulas is unsatisfiable then already some finite subset of that set must be unsatisfiable. We will see in Chapter 3 that compactness still holds in more powerful logics.

As already mentioned, soundness and completeness are desirable properties of a logic. Soundness, in fact, is crucial for any logic. Apart from providing harmony between syntax, deductive system and semantics, these two properties also save a lot of work, in translating between theorems and tautologies. In some sense theorems and tautologies are essentially the same.

Propositional logic is perhaps the simplest meaningful setting in which soundness and completeness can be studied. Predicate logic, as introduced in Chapter 3, is still complete, but the proof is beyond the scope of these lectures. Beyond that, more powerful logics quickly become incomplete due to Gödels famous results. Large parts of mathematics are based on logics, in which some tautologies cannot be derived with any sound deductive system one might conceive.

Gödel's theorems shattered a dream shared by many logicians a century ago (which was rather a nightmare in the eyes of many other mathematicians): that some powerful logic might be able to model the whole of mathematics, and that some powerful deductive system associated with it might be able to derive all true statements of mathematics algorithmically or even automatically within this setting. By Gödel's incompleteness theorems, this *logicistic approach* is impossible in general, yet by soundness and completeness of propositional logic, we can even decide any theorem of propositional logic by using truth tables in this particular small world.

2.10 Modelling Examples

This section shows two modelling examples based on propositional logic. Given the limited expressivity of this formalism, it is quite surprising how many non-trivial systems and task can nevertheless be encoded and solved, one way or another. The examples presented—Sudoku and graph colouring—may seem artificial, but it is not easy to come up with real world engineering examples that can be described on a few pages.

2.10.1 Sudoku

An encoding in predicate logic can be found in my book on *Modelling Computing Systems*. The Sudoku puzzle is based on a 9×9 grid that is divided into nine disjoint 3×3 subgrids called regions. The task is to label each cell (i, j) of the grid with a number $k \in \{1, \dots, 9\}$ such that each row, column and region contains each number precisely once, starting from an initial configuration of labels for some cells.

We write $N = \{1, \dots, 9\}$ and use the propositional variable λ_{ij}^k to indicate that cell (i, j) is labelled with number k. We write $\bigwedge_{i \in N} P_i = P_1 \wedge \cdots \wedge P_9$, and similarly for disjunction and other index conditions. Then $\bigwedge_{i \in N} \lambda_{ij}^k$, for instance, means that every element in column j has colour k and $\bigvee_{j \in N} \lambda_{ij}^k$ that some element in row i has colour k. We an thus express the constraints of Sudoku as follows.

$$
\begin{aligned}
C_1 &= \bigwedge_{i,j,k_1,k_2 \in N, k_1 < k_2} \neg(\lambda_{ij}^{k_1} \wedge \lambda_{ij}^{k_2}) && \text{(no cell contains more than one number)},\\
C_2 &= \bigwedge_{i,k \in N} \bigvee_{j \in N} \lambda_{ij}^k && \text{(every row contains every number)},\\
C_3 &= \bigwedge_{j,k \in N} \bigvee_{i \in N} \lambda_{ij}^k && \text{(every column contains every number)},\\
C_4 &= \bigwedge_{k \in N} \bigwedge_{l,m \in \{0,1,2\}} \bigvee_{i,j \in \{1,2,3\}} \lambda_{(3l+i)(3m+j)}^k && \text{(every region contains every number)}.
\end{aligned}
$$

The encoding of an initial configuration C_5 is straightforward. It is just a conjunction of variables λ_{ij}^k for fixed numbers i, j and k.

Then a particular Sudoku puzzle has a solution if and only if $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$ is satisfiable, that is, there exists a valuation v of the $9^3 = 729$ variables λ_{ij}^k such that

$$
[\![C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5]\!]_v = 1.
$$

Strictly speaking, each of the C_i must be translated into a proper propositional formula, which too large to be shown. The resulting truth table has 2^{729} rows, yet modern SAT-solvers can solve Sudoku puzzles very quickly, and there are even faster methods around!

2.10.2 Graph Colourings

A colouring of a graph is a labelling of its vertices with colours such that no adjacent vertices have the same colour. The *k-colouring problem* asks whether a given graph can be coloured with k colours.

Remember that a graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, such that $E \subseteq V \times V$. We assume that $(v, v) \in E$ for all V, that is, there are no loops, and that $(v, w) \in E$ implies that $(w, v) \in E$, that is, edges have no direction. Two vertices v and w are *adjacent* if $(v, w) \in E$, that is, there is an edge between them. A *k-colouring* of G is a labelling of the vertices of G with a "colour" from the set of numbers $C = \{1, \dots, k\}$ such that all adjacent vertices have different colour.

Let the propositional variable λ_v^c say that vertex v has colour c. We can then express the k-colouring property as follows:

$$\begin{aligned}
C_1 &= \bigwedge_{v \in V} \bigvee_{c \in C} \lambda_v^c && \text{(every vertex has at least one colour)},\\
C_2 &= \bigwedge_{v \in V} \bigwedge_{c_1, c_2 \in C, c_1 < c_2} \neg(\lambda_v^{c_1} \wedge \lambda_v^{c_2}) && \text{(every vertex has at most one colour)},\\
C_3 &= \bigwedge_{(v,w) \in E} \bigwedge_{c \in C} \neg(\lambda_v^c \wedge \lambda_w^c) && \text{(no pair of adjacent vertices has the same colour)}.
\end{aligned}$$

Thus that a given graph G is k-colourable if and only if exists a valuation v such that

$$[\![C_1 \wedge C_2 \wedge C_3]\!]_v = 1.$$

Once again, SAT solvers can be used to decide the k-colour problem. To this end, each of the C_i must be translated into a proper propositional formula. For $V = \{v_1, v_2\}$ and $C = \{1, 2\}$ for instance, $C_1 = (\lambda_{v_1}^1 \vee \lambda_{v_1}^2) \wedge \lambda_{v_2}^1 \vee \lambda_{v_2}^2)$. The formulas grow quite quickly in $|V|$ and k.

The 2-colouring problem asks whether a given graph is *bipartite*. It can be decided in linear time in the size of the graph, for instance by depth-first search. For $k \geq 3$, the problem is NP-complete. Nevertheless, it can be solved for small graphs. The minimal number of colours needed for colouring a given graph is known as the *chromatic number* of that graph. The chromatic number can be obtained by computing solutions the k-colouring problem as part of a binary search procedure. Obviously, one can always colour a graph with $|V|$ colours.

The encodings of the Sudoku puzzle and the vertex colouring problem look quite similar, and in fact one can encode the former as an instance of the latter. Other scheduling problems, including timetablings or taxi allocations to customers, can be modelled as graph colouring problems as well.

2.10.3 König's Lemma

König's lemma is an important graph theoretical result that has many applications in mathematics and computer science. A tree is *infinite* if it has infinitely many nodes and *finitely branching* if every node has only finitely many children. The proof presented is a direct application of the compactness theorem.

Theorem 2.57 (König's Lemma)**.** *Every infinite finitely branching tree has an infinite branch.*

Proof. Fix an infinite tree T. Its level can be defined recursively: $S_0 = \{c_0\}$, where c_0 is the root of T, and $S_{n+1} = \{c \mid \text{the parent of } c \text{ is in } S_k\}$. Each level S_k is of course finite, and non-empty because T is infinite. It follows that T has countably many nodes. A branch of T is a path from the root of T to one of its leaves, hence a set of nodes that are related by the predecessor or parent relation $<$ on T: $c < c'$ holds if c is the parent of c'.

For each $c \in T$ let π_v be a propositional variable that denotes that node c is on path π. The following infinite set of propositional formulas models an infinite branch in T.

$$C_1 = \bigwedge_k \bigvee_{c \in S_k} \pi_v \qquad \text{(each level has at least one node in } \pi)$$

$$C_2 = \bigwedge_k \bigwedge_{c,c' \in S_k, c \neq c'} \neg(\pi_c \wedge \pi_{c'}) \qquad \text{(each level has at most one node in } \pi)$$

$$C_3 = \bigwedge_{c,c' \in V, c < c'} \pi_{c'} \rightarrow \pi_c \qquad (\pi \text{ is closed with respect to predecessors})$$

$C_1 \wedge C_2$ guarantees that π crosses each level of T precisely once. In conjunction with C_3 it follows that the nodes that satisfy $v(\pi) = 1$ respect the predecessor relation. They form indeed a branch which extends from level to level ad infinitum. It remains to check that the infinite set $\Gamma = C_1 \wedge C_2 \wedge C_3$ is satisfiable.

By compactness it suffices to show that every finite subset of Γ is satisfiable. So let Γ_0 be a fixed but arbitrary finite subset of Γ and let n be the maximal layer of T some propositional variable in Γ_0 refers to. Then there exists a finite set $\Gamma_n \supseteq \Gamma_0$, which is defined like Γ, but with formulas C_1, C_2 and C_3 ranging up to level n only. The set Γ_n thus describes a branch up to level n; it is finite and clearly satisfiable, hence so is its subset Γ_0. $\square$

Hence the compactness theorem yields a simple proof of König's lemma, but König's lemma can also be proved by other means. In that case, the compactness theorem for propositional logic, and even its companion for predicate logic (Theorem 3.49 in Section 3.7), can be proved directly by using König's lemma, which underpins the importance of this lemma.

2.11 List of Inference Rules of Natural Deduction

2.11.1 Intuitionistic Rules

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi}\wedge I \qquad\qquad \frac{\varphi \wedge \psi}{\varphi}\wedge E_l \qquad\qquad \frac{\varphi \wedge \psi}{\psi}\wedge E_r$$

$$\frac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \to \psi}\to I \qquad\qquad \frac{\varphi \to \psi \qquad \varphi}{\psi}\to E$$

$$\frac{}{\top}\top I \qquad\qquad \frac{\bot}{\varphi}\bot E$$

$$\frac{\varphi}{\varphi \vee \psi}\vee I_l \qquad\qquad \frac{\psi}{\varphi \vee \psi}\vee I_r \qquad\qquad \frac{\varphi \vee \psi \qquad \begin{array}{c}[\varphi]\\ \vdots\\ \chi\end{array} \qquad \begin{array}{c}[\psi]\\ \vdots\\ \chi\end{array}}{\chi}\vee E$$

$$\frac{\begin{array}{c}[\varphi]\\ \vdots\\ \bot\end{array}}{\neg\varphi}\neg I \qquad\qquad \frac{\varphi \qquad \neg\varphi}{\bot}\neg E$$

2.11.2 Classical Rules

$$\frac{}{\varphi \vee \neg\varphi}\,\mathrm{lem} \qquad\qquad \frac{\begin{array}{c}[\neg\varphi]\\ \vdots\\ \bot\end{array}}{\varphi}\,\mathrm{pbc} \qquad\qquad \frac{\neg\neg\varphi}{\varphi}\neg\neg E$$

2.12 Propositional Logic with Isabelle/HOL

In the lectures I show how natural deduction proofs can be executed on a machine. The software that allows me to do this is the interactive theorem prover or proof assistant Isabelle/HOL. HOL stands for *Higher-Order Logic*, which is a logic much more powerful than propositional logic and in which much of mathematics can be formalised. In fact, many mathematical concepts from fields including algebra, analysis, probability theory, or topology have been formalised and many facts from these fields have been proved.

Using Isabelle requires a strong background in logic and mathematics, and it takes some time to use this tool fluently. However, there is no time to introduce Isabelle adequately in this course and I do not expect you to use it yourself.

Nevertheless, with some patience, even beginners can do some simple proofs in propositional logic with Isabelle, either step by step by natural deduction or even fully automatically by using Isabelle's built in solvers and proof search methods.

It might be a surprising and even frustrating experience to see how easily tools like Isabelle can prove theorems fully automatically that you may have struggled with in the exercises. This has nothing to do with intelligence. Computers can just perform combinatorial task such as building and checking an enormous number of derivation trees very quickly. Even programming the basic algorithms that enable computers to perform such tasks is a routine exercise—after two thousand years of logic and more than fifty years of computer science—and can be done in various ways as a student project. There is often less intelligence in artificial intelligence than one might think!

If you want to try Isabelle yourself, you can download the tool here:

`https://isabelle.in.tum.de`

This web site provides a wealth of information—probably more than anyone can digest. A direct link to tutorials and reference manuals is

`https://isabelle.in.tum.de/documentation.html`

A gentle introduction to programming and proving with Isabelle can be found here:

`https://isabelle.in.tum.de/dist/Isabelle2017/doc/prog-prove.pdf`

Detailed information on the introduction and elimination rules of natural deduction used by Isabelle can be found in an older tutorial (Section 5).

`https://isabelle.in.tum.de/dist/Isabelle2017/doc/tutorial.pdf`

If you want to learn more about Isabelle, you should attend my course COM4507/6507 on *Hardware and Software Verification* —or contact Achim Bucker or me about an Isabelle project in year 3.

=== END OF WEEK FIVE (EXPECTED) ====================

Chapter 3

Predicate Logic

3.1 What is Predicate Logic?

Propositional logic, as studied in Chapter 2, takes propositional variables as basic building blocks and uses negation, conjunction, disjunction and implication to build composite propositions. Though it may be surprising how non-trivial systems properties can be encoded, one way or another, and analysed within this formalism, it is certainly not expressive enough for many applications. Even simple arguments, like the famous syllogism in Example 2.1, cannot be expressed appropriately. The internal structure of declarative sentences such as

All children love ice cream.
Some students understand implication.
No electron has positive charge.

cannot be captured. Expressions such as"for all" and "there exists", for instance, which quantify over a given set or category of entities and are therefore called *quantifiers*, cannot be represented. Reasoning based on the relationships between entities, such as

If $x \leq 7$ *and* $9 \leq y$*, then* $x \leq y$.
If $y = 2x + 6$*, then* $x = \frac{1}{2}y - 3$.

which is captured in the internal structure of propositions as well, is impossible. Propositions such as $p \wedge q \to r$ or $p \to q$ are far to coarse.

A more refined logic requires first of all a more expressive language. It should capture terms such as $2x + 6$, relations or predicates such as $x \leq 7$, and quantification in sentences such as "All x are P" or "Some x are P". Secondly, new inference rules are needed to reason with this more expressive language. Thirdly, a more fine grained semantics is required beyond mere truth values. It needs to supply objects, functions and relations that match the syntactic expressions in the language. Terms such as $2 \cdot 4 + 6$ or $9 \leq 7$, for instance, need to be evaluated in domains of discourse such as the natural numbers. Predicate logic, or *first-order logic*, as it is also called, satisfies these requirements.

A natural question then concerns the properties of predicate logic. As for propositional logic, we may ask about soundness and completeness, compactness, decidability of satisfiability, and so forth. These questions are answered in this chapter as well. Yet, like for propositional logic, we start from a foundational point of view and present the rules of natural deduction for predicate logic in a semi-formal way as universal laws of reasoning.

3.2 Natural Deduction

Natural deduction for predicate logic merely adds introduction and elimination rules for quantifiers to those of propositional logic. Hence it comes in a constructive or intuitionistic and in a classical variant. The rules for quantification have computational content, but it is more difficult to explain than for the propositional rules. Here, we therefore restrict our attention to the classical case. First of all, a new form of judgment is required to to explain the quantifier rules.

3.2.1 Parametric Judgments

At this stage we are rather unspecific about the syntax of predicate logic. A formal introduction is deferred to Section 3.4. Forming statements like "All x are P" requires that propositions depend on x in the same way functions $f(x)$ do. We write $P(x)$ to indicate that P takes x as an argument or parameter. Such propositions are no longer propositions in a strict sense—we cannot immediately ascribe a truth value to"x is human", yet they become propositions as soon as their variables have been replaced by suitable values such as "Socrates". For the time being we call such values *terms* without attempting a formal definition. Hence we can make a judgment $P(x)$ *true* after substituting any suitable term t for the variable x in P, which we indicate by $P(t)$ or $P[t/x]$. Substituting the term 5 into $P(x) = x \leq 7$, for instance, yields the proposition $5 \leq 7$. Henceforth we use the term propositions also in this loose sense.

To explain quantification, it is important to make judgments for an arbitrary value, to say that a proposition $P(t)$ is true for an arbitrary term t. Mathematicians do this all the time: Proofs that all natural numbers have a certain property often start with the sentence "Let us fix an arbitrary natural number n". One then shows that n satisfies the desired properties, and concludes that it therefore holds for all natural numbers. Of course there are no arbitrary values, yet what mathematicians mean is that no assumptions beyond their existence can be made, that is, the values are new and have not been mentioned elsewhere in the proof.

To mimic the mathematicians's trick with natural deduction, we introduce *parameters* as special kinds of terms, and consider *parametric judgments* of the form

J *for an arbitrary* a.

where a is a parameter on which J may depend. The parameter of an argument represents precisely the value about which nothing is assumed.

With a parametric judgment, we are prepared to assume that it holds for every suitable value or object in the domain we reason about. We are thus prepared to substitute any particular term t for a parameter a in a proposition P when making the parametric judgment P *true for an arbitrary a*. In other words, P *true for an arbitrary a* if and only if $P[t/a]$ *true* for every t. Parametric judgments thus allow us to reduce judgments about an entire class or category of objects to those about a fixed but arbitrary object represented by a parameter. Their relevance becomes clear when we consider universal quantification.

3.2.2 Universal Quantification

In predicate logic, if φ is a proposition and x a *variable* drawn from some suitable set, then $\forall x.\ \varphi$ is proposition. This is, in fact, a formation rule, but we do not associate any further importance to it at this stage. We often write $\forall x.\ \varphi(x)$ to indicate that the proposition φ depends on x or has x as an argument. We read $\forall x.\ \varphi$ as "Every x satisfies φ." or "Every x is a φ.". This is obviously a declarative sentence and hence a proposition.

Next we turn to the inference rules for universal quantification; its introduction and its elimination rule. As usual, we abbreviate judgments $\forall x.\ \varphi$ *true* as $\forall x.\ \varphi$.

The introduction rule defines the operational meaning of universal quantification, so we explain it first. If we know that the proposition $\varphi(a)$ is true for an arbitrary parameter a we assume nothing about, if we are prepared to make the parametric judgment $\varphi(a)$ *true for an arbitrary a*, then we are prepared to accept the judgment $\forall x.\ \varphi$ *true.* To indicate the relationship between a and x, we write $\varphi[a/x]$ in the premise of the rule.

To explain the elimination rule, we ask, as usual, what $\forall x.\varphi$ gives us in a proof. By the introduction rule, of course, we know that $\varphi[a/x]$ is true for an arbitrary parameter a. Yet, by the previous section, we also know that we can substitute any term or value t for the parameter a in φ. Hence we may conclude that $\varphi[t/x]$ holds for every term t.

In sum, we obtain the inference rules

$$\frac{\varphi[a/x]}{\forall x.\ \varphi}\,\forall I \qquad\qquad \frac{\forall x.\ \varphi}{\varphi[t/x]}\,\forall E$$

where t is any term and a a fresh parameter we know nothing about. To indicate that a whole derivation leading to the premise $\varphi[a/x]$ may depend on a, the introduction rule is sometimes written

$$\begin{array}{c} \text{—}\ a \\ \vdots \\ \dfrac{\varphi[a/x]}{\forall x.\ \varphi}\,\forall I \end{array}$$

Additional intuition for the introduction rule for universal quantification can be drawn from a comparison with propositional logic. Consider, for instance, the finite set

$$S = \{1, 2, 3, 4, 5\}$$

as our domain of discourse. Then the proposition "All elements of S have one digit" is true if and only if 1 and 2 and 3 and 4 and 5 each have one digit, that is,

$$\forall x.\ D(x) = D[1/x] \wedge D[2/x] \wedge D[3/x] \wedge D[4/x] \wedge D[5/x],$$

and with $(\wedge I)$ we could conclude this conjunction from the premises $D[1/x]$, $D[2/x]$, $D[3/x]$, $D[4/x]$ and $D[5/x]$. The introduction rule for universal quantification would thus be a conjunction-style rule with five premises. By analogy, a universal quantification over an infinite domain such as $\mathbb{N}$ would thus correspond to an infinite conjunction; its introduction rule would be a conjunction-style inference rule with infinitely many premises. It is neither desirable that inference rules depend on properties of the underlying domain, nor feasible that they have infinitely many premises. How could we even make infinitely many judgments? Instead, the introduction rule $(\forall I)$ uses the mathematicians' trick mentioned in the previous section and reduces the consideration of infinitely many concrete values to that of one single arbitrary one—a parameter.

Next we present two simple examples that show the introduction and elimination rule for universal quantification at work.

Our first example derives the judgment about Socrates made in Example 2.1.

Lemma 3.1. *$\forall x.\ H(x) \to M(x), H(s) \vdash M(s)$.*

Proof.

1.	$\forall x.\ H(x) \to M(x)$	hyp
2.	$H(s)$	hyp
3.	$H(s) \to M(s)$	$\forall E$, 1
4.	$M(s)$	$\to E$, 3,2

□

The following lemma shows how the parametric judgment leading to $(\to I)$ is made in a linear proof. As for hypothetical judgments, we put parametric judgments into boxes in order to manage the "locality" of the parameter declaration with respect to the inference made. Nevertheless, hypothetical and parametric judgments should not be confused.

Lemma 3.2. *$\vdash \forall x.\ P(x) \to Q(x), \forall x.\ P(x) \vdash \forall x.\ Q(x)$.*

Proof.

1.	$\forall x.\ P(x) \to Q(x)$	hyp
2.	$\forall x.\ P(x)$	hyp
3.	a	
4.	$P(a) \to Q(a)$	$\forall E$, 1
5.	$P(a)$	$\forall E$, 2
6.	$Q(a)$	$\to E$, 5,4
7.	$\forall x.\ Q(x)$	$\forall I$, 3-6

The parameter a must be introduced early on in this proof. At the time of its introduction, it had to be fresh. Thus we had to eliminate universal quantifiers after its introduction. The rule $(\forall E)$ tells us that we can substitute any term for the variable we quantified over in its conclusion—hence in particular a. As mentioned in Section 3.2.1, parameters are special kinds of terms. □

The introduction and elimination rules for universal quantification satisfy the principle of harmony. The reduction property of the introduction step before the elimination rules should, as always, be evident from the explanation of $(\forall E)$. The harmony proofs are similar to those for conjunction, but slightly complicated by the presence of parameters. They are beyond the scope of these lectures.

The final example—or rather non-example—of this section highlights once again the use of parameters in the introduction rule for universal quantification.

Example 3.3. Suppose we want to prove $\vdash \forall x \forall y.\ P(x) \to P(y)$. This should clearly be impossible: if $P(x)$ stands for $x \leq 7$, then $\forall x \forall y.\ x \leq 7 \to y \leq 7$ is clearly false—just take $x = 6$ and $y = 8$. So let us see how natural deduction deals with this situation.

1.	a	
2.	b	
3.	$P(a)$	hyp
4.	$\vdots$	?
5.	$P(b)$	?
6.	$P(a) \to P(b)$	$\to I$ ...
7.	$\forall y.\ P(a) \to P(y)$	$\forall I$ ...
8.	$\forall x \forall y.\ P(x) \to P(y)$	$\forall I$ ...

The workflow of this proof is deterministic and bottom-up. There is only one introduction rule applicable at each stage. The last proof step is $(\to I)$, after which we are left with proving $P(b)$ from the hypothesis $P(a)$. This is clearly impossible with the information we have. In particular we cannot argue that a and b must be equal. These two parameters were enforced by the two previous applications of $(\forall I)$. We could not pick a again in the second proof step because it was no longer fresh.

Of course this discussion does not prove that we cannot finish the proof—only a soundness result would tell us that the counterexample rules out a proof. But, from a pragmatic point of view, which inference rules could we have used to close the gap? □

3.2.3 Existential Quantification

As for universal quantification, $\exists x.\ \varphi$ is a proposition whenever φ is. We read $\exists x.\ \varphi$ as "Some x satisfies φ" or "There exists an x such that φ".

The introduction rule for existential quantification is very straightforward. If we can find a witness t for which $\varphi(t)$ is true—a particular term or value for which we can prove $\varphi(t)$—then we know that φ holds of some x of the appropriate type, and therefore $\exists x.\ \varphi$. To indicate the correspondence between x and t, we write $\varphi[t/x]$ in the premise.

To explain the elimination rule we should ask once more what $\exists x.\ \varphi$ gives us in a proof. By the introduction rule, the answer is of course a proof of $\varphi[t/x]$ for some witness t. Now suppose that we have a hypothetical proof of some conclusion ψ from an arbitrary parameter a in place of x in φ. That is, a hypothetical proof of ψ from $\varphi[a/x]$ which is at the same time parametric. Then we can use the proof of $\varphi[t/x]$ to extend the hypothetical parametric proof of ψ to a proof of ψ that depends neither on a parameter nor on the hypothesis, that is, we can kill the hypothesis $\varphi[a/x]$.

We thus obtain the following rules:

$$\frac{[\varphi[t/x]]}{\exists x.\ \varphi}\ \exists I \qquad\qquad \frac{\exists x.\ \varphi \qquad \begin{array}{c}[\varphi[a/x]]\\ \vdots \\ \psi\end{array}}{\psi}\ \exists E$$

where t is any term and a a fresh parameter we know nothing about.

A second way of explaining the elimination rule for existential quantification can be based one again on the consideration of a finite domain. So let again

$$S = \{1, 2, 3, 4, 5\}$$

be the domain of discourse and consider the proposition "Some number in S is even". It is true if and only if either 1 or 2 or 3 or 4 or 5 is even. In this case, existential quantification reduces to a disjunction and the elimination rule to a disjunction elimination rule with six premises, five of which are hypothetical proofs. These could be discharged after the formula ψ has been proved. As in the case of $(\forall I)$, an arbitrary existential quantification thus amounts to an infinite disjunction and the corresponding case analysis would force us to perform infinitely many conditional proofs, which is impossible. Hence again, the infinitely many hypothetical proofs for concrete values are replaced by one single hypothetic parametric proof in a, an arbitrary value about which we cannot assume anything.

As for universal quantification, the introduction and elimination rules are in harmony; the reduction part has already been explained for the elimination rule. We do not show any details.

Next we present some example derivations.

Lemma 3.4. $\forall x.\ \varphi \vdash \exists x.\ \varphi$.

Proof.

1.	$\forall x.\ \varphi$	hyp
2.	$\varphi[t/x]$	$\forall E$, 1
3.	$\exists x.\varphi$	$\exists I$, 2

□

The next example can be seen as a generalised version of the Socrates example.

Lemma 3.5. *$\forall x.\ P(x) \to Q(x), \exists x.\ P(x) \vdash \exists x.\ Q(x)$.*

Proof.

1.	$\forall x.\ P(x) \to Q(x)$	hyp
2.	$\exists x.\ P(x)$	hyp
3.	a	
4.	$P(a)$	hyp
5.	$P(a) \to Q(a)$	$\forall E$, 1
6.	$Q(a)$	$\to E$, 3,4
7.	$\exists x.\ Q(x)$	$\exists I$, 5
8.	$\exists x.\ Q(x)$	$\exists E$, 2,3-6

□

Lemma 3.6. *$\exists x.\ P(x), \forall x \forall y.\ (P(x) \to Q(y)) \vdash \forall y.\ Q(y)$.*

Proof.

1.	$\exists x.\ P(x)$	hyp
2.	$\forall x \forall y.\ (P(x) \to Q(y))$	hyp
3.	a	
4.	b	
5.	$P(b)$	hyp
6.	$\forall y.\ (P(b) \to Q(y))$	$\forall E$, 2
7.	$P(b) \to Q(a)$	$\forall E$, 5
8.	$Q(a)$	$\to E$, 4,6
9.	$Q(a)$	$\exists E$, 1,4-7
10.	$\forall y.\ Q(y)$	$\forall I$, 3-8

□

3.2.4 Equality

Equality is one of the most important operations in mathematics and computing. This is why it has special status, and we supply natural deduction rules for it as well. The introduction rule for equality is very simple.

$$\frac{}{\text{t=t}} = I$$

where t is any term or value. We know that each object is equal to itself, and this knowledge does not need any hypothesis. The introduction rule for equality is also know as the axiom of *reflexivity* for equality.

To explain the elimination rule, we ask an equation $t_1 = t_2$ gives us in a proof. The obvious answer is that we can substitute t_2 for t_1 in any proposition φ in which t_1 occurs.

$$\frac{t_1 = t_2 \qquad \varphi[t_1/x]}{\varphi[t_2/x]} = E$$

This rule is so famous that it has another name, too: *Leibniz's law*, after the famous 17th century philosopher—and logician. The intuition behind Leibniz's law is as follows: if t_1 and t_2 are equal, then we cannot distinguish them in any context; they behave precisely in the same way, and one can always replace one by the other.

Equality has two important properties as a binary relation beyond reflexivity: it is *symmetric* ($t_1 = t_2$ implies $t_2 = t_1$) and *transitive* ($t_1 = t_2$ and $t_2 = t_3$ imply $t_1 = t_3$). These properties are derivable.

Lemma 3.7.

1. *Equality is symmetric:* $t_1 = t_2 \vdash t_2 = t_1$.

2. *Equality is transitive:* $t_1 = t_2 . t_2 = t_3 \vdash t_1 = t_3$.

Proof.

1. Symmetry:

1.	$t_1 = t_2$	hyp
2.	$t_1 = t_1$	$= I$
3.	$(x = t_1)[t_1/x]$	rewriting 2
4.	$(x = t_1)[t_2/x]$	$= E$, 1,3
5.	$t_2 = t_1$	rewriting 4

2. Transitivity:

1.	$t_1 = t_2$	hyp
2.	$t_2 = t_3$	hyp
3.	$(t_1 = x)[t_2/x]$	rewriting 1
4.	$(t_2 = x)[t_3/x]$	$= E$, 2,3
5.	$t_1 = t_3$	rewriting 4

□

The introduction and elimination rules for equality satisfy the principle of harmony, which is left as an exercise.

This completes the natural deduction formalism for predicate logic. Once more, there is an intuitionistic or constructive and a classical variant, depending on whether or not the law of excluded middle, the rule for proof by contradiction or the rule of double negation are assumed. The complete list of inference rules can be found in Section 3.9.

All rules of natural deduction satisfy the principle of harmony (except perhaps those for truth and falsity). This leads to the pleasant normal form property that all derivations can split into a phase with only elimination rules followed by a second phase with only introduction rules, at least in the constructive case. Trying proofs with elimination rules from the top and introduction rules from the bottom remains generally a good strategy, though it may not always be desirable.

3.3 Helpful Properties

This section simply lists some helpful properties of quantified formulas. Most of them are taken from Huth and Ryan's book, where you can also find some proofs.

Lemma 3.8. *The following De Morgan laws hold.*

1. $\neg\forall x.\ \varphi \dashv\vdash \exists x.\ \neg\varphi$.
2. $\neg\exists x.\ \varphi \dashv\vdash \forall x.\ \neg\varphi$.

Lemma 3.9.

1. $\forall x \forall y.\ \varphi \dashv\vdash \forall y \forall x.\ \varphi$.
2. $\exists x \exists y.\ \varphi \dashv\vdash \exists y \exists x.\ \varphi$.
3. $\exists x \forall y.\ \varphi \vdash \forall y \exists x . \varphi$.
4. $\forall x.\ \varphi \wedge \forall x.\ \psi \dashv\vdash \forall x.\ (\varphi \wedge \psi)$.
5. $\exists x.\ \varphi \vee \exists x.\ \psi \dashv\vdash \exists x.\ (\varphi \vee \psi)$.

Lemma 3.10. *Suppose that x is not free in ψ.*

1. $(\forall x.\ \varphi) \wedge \psi \dashv\vdash \forall x.\ (\varphi \wedge \psi)$.
2. $(\forall x.\ \varphi) \vee \psi \dashv\vdash \forall x.\ (\varphi \vee \psi)$.
3. $(\exists x.\ \varphi) \wedge \psi \dashv\vdash \exists x.\ (\varphi \wedge \psi)$.
4. $(\exists x.\ \varphi) \vee \psi \dashv\vdash \exists x.\ (\varphi \vee \psi)$.
5. $\psi \to \forall x.\ \varphi \dashv\vdash \forall x.\ (\psi \to \varphi)$.

6. $(\forall x.\ \varphi) \to \psi \dashv\vdash \exists x.\ (\varphi \to \psi)$.

7. $\psi \to \exists x.\ \varphi \dashv\vdash \exists x.\ (\psi \to \varphi)$.

8. $(\exists x.\ \varphi) \to \psi \dashv\vdash \forall x.\ (\varphi \to \psi)$.

As to equality, it is helpful to use the rules derived in Lemma 3.7:

$$\frac{t_1 = t_2}{t_2 = t_1}\,\text{sym} \qquad \frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_2}\,\text{trans}$$

Another useful derived rule is the substitution rule

$$\frac{t_1 = t_2}{s[t_1/x] = s[t_2/x]}\,\text{subst}$$

3.4 Syntax

As in the case of propositional logic, we now consider predicate logic as a mathematical theory.

3.4.1 The Language of Predicate Logic

Definition 3.11 (Alphabet)**.** An *alphabet* for predicate logic consists of the following:

1. A set (countably infinite) set $\mathcal{V}$ of logical *variables*;
2. a set $\mathcal{F} = \{f_1, \dots f_n\}$ of *function symbols*, each with a fixed arity[1];
3. a set $\mathcal{P} = \{P_1, \dots, P_n\}$ of *predicate symbols*, each with a fixed arity;
4. the special predicate symbol $=$ of *equality*;
5. the *connectives* $\bot$, $\top$, $\neg$, $\wedge$, $\vee$, $\to$, $\forall$, $\exists$; item auxiliary symbols (,), etc.

The non-logical symbols (in $\mathcal{F}$ and $\mathcal{P}$) of an alphabet are called its *signature*. Equality counts as a special logical symbol.

Function symbols of arity 0 take no arguments. They are called *constant symbols*. We usually write $c, c_0, c_1, \dots$ for constant symbols, and Σ for signatures. Signatures consisting solely of relation symbols are called *relational signatures*. Those consisting solely of function symbols are called *algebraic signatures*.

Example 3.12.

1. The signature of arithmetic is $\Sigma_A = \{+, \cdot, 0, 1\}$; $+$ and $\cdot$ are binary function symbols, 0 and 1 are constant symbols.

[1]The arity of a function or relation symbol is the number of parameters it can take

2. The signature of boolean algebras is $\Sigma_{BA} = \{\sqcap, \sqcup, -, 0, 1\}$; $\sqcap$ and $\sqcap$ are binary function symbols, $-$ is a unary function symbol, 0 and 1 are constant symbols.

3. The signature of graphs is $\Sigma_G = \{E\}$, where E is a binary relation symbol.

□

Definition 3.13 (Terms)**.** Let Σ be a signature. The set $\mathcal{T}_\Sigma$ of Σ-*terms* is defined by the following grammar. For $x \in \mathcal{V}$, $f \in \mathcal{F}$ of arity m,

$$\mathcal{T}_\Sigma ::= x \mid f(\mathcal{T}_\Sigma, \ldots, \mathcal{T}_\Sigma).$$

Terms are thus defined recursively, like propositional formulas in Definition 2.23. The basic building blocks are variables and constant symbols. Composite terms are obtained by applying function symbols of arity > 0 to terms that have already been constructed, in such a way that their arity matches the number of terms it is applied to.

Example 3.14. Terms correspond to expressions in programming languages. They can be evaluated—at least partially if they contain variables—but not be true or false in general.

1. Extend the signature Σ_A of arithmetic with constant symbols $2, 3, 4, \ldots$ for each natural number $\mathbb{N}$. Let $x, y \in \mathcal{V}$. Then $3 + 5 \cdot 7$ is a term; its value or interpretation is the number 38. Another term is $x + 5 \cdot y$, but it cannot be evaluated to a natural number.

2. Extend the Signature Σ_{BA} of boolean algebra with constant symbols a and b. Then $a \sqcap -a$ is a term, and so is $a \sqcup (b \sqcap -a)$. These terms evaluate to elements of a boolean algebra ($a \sqcap -a$ usually becomes the least element of a boolean algebra), but not to true or false.

3. Consider the signature $\{\mathit{MotherOf}, \mathit{Alice}, \mathit{Betty}\}$, where *MotherOf* is a unary function symbol and *Alice* and *Betty* are constant symbols. Then $\mathit{MotherOf}(\mathit{Alice})$ is a term. Again, it does not evaluate to a truth value, but to another value, for instance to Betty, the person represented by the constant symbol *Betty*.

How precisely terms are evaluated will be explained in Section 3.6. □

The recursive definition of terms, like that of propositional formulas, is essentially that of a tree data type, whose leaves are labelled by constants and variables, and whose inner nodes are labelled by functions of arity > 0.

Example 3.15. The arithmetic expression $3 + 5 \cdot 7$ from Example 3.14(1) corresponds to the labelled tree

$$\begin{tikzcd}
 & + \arrow[dl, dash] \arrow[dr, dash] & & \\
3 & & \cdot \arrow[dl, dash] \arrow[dr, dash] & \\
 & 5 & & 7
\end{tikzcd}$$

Evaluating it step by step in bottom-up fashion in the natural numbers yields

$$\begin{tikzcd} & + \arrow[dl, dash] \arrow[dr, dash] & \\ 3 & & 35 \end{tikzcd} \qquad \text{and then} \qquad 38.$$

Once again, a precise notion of evaluation, and a formal semantics of terms, is introduced in Section 3.6. □

Due to their recursive definition, properties of terms such as subterms, occurrences or sets of variables that occur in them, can be defined recursively, like for propositional formulas in Section 2.8.1. We leave these as exercises. Terms which do not have any variable as a subterm are called *ground terms*.

Definition 3.16 (Formulas)**.** Let Σ be a signature. The set Φ_Σ of Σ-*formulas* of predicate logic is defined by the following grammar. For all $t_1, \ldots t_n \in \mathcal{T}_\Sigma$ and $P \in \mathcal{P}$ of arity n,

$$\Phi_\Sigma ::= \bot \mid \top \mid t_1 = t_2 \mid P(t_1, \ldots, t_n) \mid (\neg\Phi_\Sigma) \mid (\Phi_\Sigma \wedge \Phi_\Sigma) \mid (\Phi_\Sigma \vee \Phi_\Sigma) \mid (\Phi_\Sigma \to \Phi_\Sigma) \mid (\forall x.\ \Phi_\Sigma) \mid (\exists x.\ \Phi_\Sigma).$$

- Formulas of the form $\bot$, $\top$, $t_1 = t_2$ and $P(t_1, \ldots, t_n)$ are *atomic*;
- all other formulas are *composite*.

An alphabet together with a concrete syntax for terms and formulas is often called a *language* for predicate logic.

The definition of formulas of predicate or first-order logic allows quantification over variables only. Quantification over function or predicate symbols is not supported. The induction principle for natural numbers, for instance, cannot be expressed. It requires a quantification over predicate symbols:

$$\forall P.\ (P(0) \wedge \forall k.\ P(k) \to P(k+1)) \to \forall n.\ P(n)).$$

A natural language example for a quantification over a predicate is "Some of my friends don't like each other". This sentence does not simply express that a fixed number of my friend do not like each each other, but that some some *group* of friends mutually dislikes each other.

Quantification over function and predicate symbols is called *second-order* quantification; logics that allow it are called *second-order* or *higher-order* logics. Quantification over variables ranges only over individuals of an universe of discourse and it is called *first-order*. This explains why predicate logic is called *first-order logic*.

Definition 3.17 (Operator Precedence)**.** Among the connectives $\neg$, $\wedge$, $\vee$, $\to$, $\forall x$ and $\exists x$,

1. $\neg$, $\forall x$ and $\exists x$ bind equally strongly, and more strongly than the binary connectives,
2. $\wedge$ and $\vee$ have equal precedence, and they bind more strongly than $\to$.

In addition to using the precedence to save brackets, we usually write $\forall x \exists y.\ \varphi$ instead of $\forall x.\ \exists y.\ \varphi$ and likewise for other quantifier combinations. Another common abbreviation is $\forall x \in S.\ \varphi$ for $\forall x.\ (x \in S \to \varphi)$ and likewise for existential quantification.

Example 3.18.

1. *Loves* is a predicate symbol of arity two—it can take two values. *Alice* and *Bob* are constant symbols, and therefore terms. The atomic formula

$$\mathit{Loves}(\mathit{Alice}, \mathit{Bob})$$

states that Alice loves Bob. This declarative sentence can be true or false.

2. For a variable $x \in \mathcal{V}$, $\mathit{Loves}(x, \mathit{Bob})$ and $\mathit{Loves}(\mathit{Alice}, x)$ are atomic formulas as well, but these do not correspond immediately to declarative sentences. We need to supply a value for x before we can tell whether the formula is true or false.

3. Another way to turn $\mathit{Loves}(x, \mathit{Bob})$, $\mathit{Loves}(\mathit{Alice}, x)$ and similar sentences into declarative sentences is to add quantifiers.

   (a) $\exists x.\ \mathit{Loves}(x, \mathit{Bob})$ means that somebody loves Bob.
   (b) $\exists x.\ \mathit{Loves}(\mathit{Alice}, x)$ means that Alice loves someone.'
   (c) $\neg\exists x.\ \mathit{Loves}(x, \mathit{Bob})$ means that nobody loves Bob.'
   (d) $\forall x.\ \mathit{Loves}(x, \mathit{Alice})$ means that everybody loves Alice.'
   (e) $\neg\forall x.\ \mathit{Loves}(x, \mathit{Bob})$ means that not everybody loves Bob or, equivalently that somebody does not love Bob.
   (f) $\forall x \exists y.\ \mathit{Loves}(x, y)$ means that verybody loves somebody.'
   (g) $\exists x \forall y.\ \mathit{Loves}(x, y)$ means that somebody loves everybody.

□

Example 3.19. Consider the signature $\Sigma_{OA} = \Sigma_A \cup \{\leq\}$ of *ordered arithmetic*, extended once again by constant symbols representing the natural numbers. Obviously, $\leq$ is a binary predicate symbol. Hence the atomic formulas of the language of ordered arithmetic are $\bot$, $\top$ as well as $t_1 = t_2$ and $t_1 \leq t_2$ for arithmetic terms t_1 and t_2. Examples are

$$3 + 5 = 8, \qquad 7 + 2 \leq 3, \qquad x + 4 \cdot y = 21.$$

$3 + 6 \neq 10$, which is just an abbreviation for $\neg(3 + 6 = 10)$, is an example of composite formula in the language of arithmetic. □

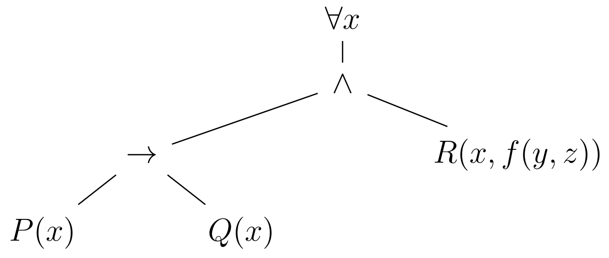
Example 3.20. In a simple programming language, predicates occur in the form of tests in conditionals or loops. A grammar for a simple while language might look like

$$\mathit{Prog} ::= x := t \mid \mathit{Prog}; \mathit{Prog} \mid \textbf{if } p \textbf{ then } \mathit{Prog} \textbf{ else } \mathit{Prog} \mid \textbf{while } p \textbf{ do } \mathit{Prog},$$

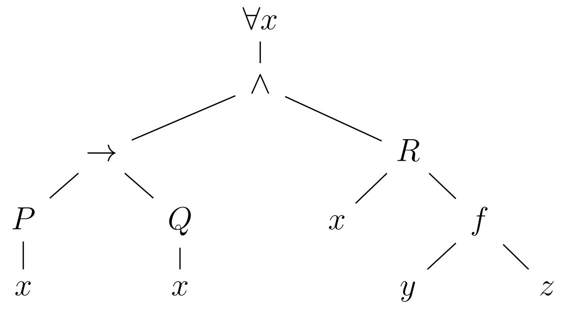
where x is a program variable, t an arithmetic term and p an arithmetic formula. This, of course, has little to do with predicate logic; the example simply shows how logical syntax can appear in programming language definitions, and is thus relevant to programming languages and computer science. $\square$

Formulas can be represented by parse trees as well. In this case, leaves are labelled by atomic formulas, and inner nodes by the connectives of predicate logic.

Example 3.21. The parse tree of the formula $\forall x.\ ((P(x) \to Q(x)) \land R(x, f(y, z)))$ is



It is often preferable do show combined parse trees, expanding the term structure beyond atomic formulas. This yields



$\square$

The following example compares the use of function and predicate symbols in translations.

Example 3.22. The sentence "Every son of my father is my brother" can be translated into predicate logic as

$$\forall x \forall y.\ (F(x, m) \land S(y, x) \to B(y, m)) \qquad \text{and} \qquad \forall x.\ (S(x, f(m)) \to B(x, m).$$

The difference is that the first formalisation uses a binary predicate symbol F for the father relation, whereas the second one uses a unary function symbol f. This is possible, because the relation of fatherhood is indeed functional: Every person has exactly one father, at least in principle. By contrast, the relations "is a son of" and "is a brother of" are not functional in general. While the syntax definitions of predicate logic do not allow us to nest predicate symbols, we can of course use a function symbol underneath a predicate symbol in a formula. Both formalisations are certainly correct. The second one is perhaps more concise, because it adds further information about kinship relations into the formalisation. $\square$

3.4.2 Free and Bound Variables

Quantifiers provide *scope* to variables in formulas in the same way local variables have a scope inside a block of code. We say that a variable in a formula is *bound* by a quantifier, or more formally, that an *occurrence* of a variable in a formula is bound.

Definition 3.23 (Occurence)**.**

1. The function $V : \mathcal{T} \to \mathcal{P}(\mathcal{V})$ defined by

$$\begin{aligned} V(x) &= \{x\}, \quad \text{if } x \in \mathcal{V}, \\ V(c) &= \emptyset, \quad \text{if } c \text{ is a constant}, \\ V(f(t_1, \ldots, t_n)) &= V(t_1) \cup \cdots \cup V(t_n), \quad \text{if } f \in \mathcal{F} \text{ has arity } n \text{ and } t_1, \ldots, t_n \in \mathcal{T}, \end{aligned}$$

   computes the variables that occur in a given term.

2. A variable x *occurs* in a term t if $x \in V(t)$.

The extension of V to a function $V : \Phi \to \mathcal{P}(\mathcal{V})$ is straightforward, and left as an exercise. With an *occurrence* of a variable x in a formula φ we mean a particular subformula x of φ, or particular a leaf of a parse tree of φ that is labelled by x. Hence a variable may occur several times in a formula. A formula in which all occurrences of variables are bound is called *closed* or a *sentence*

Example 3.24. The variable x occurs thrice in $(\forall x.\ ((P(x) \to Q(x)) \wedge R(x, f(y, z)))$, the variables y and z occur once. □

Definition 3.25 (Bound and Free Variables)**.**

1. An occurrence of a variable x in a formula φ is *bound* if x occurs in a subformula $\forall x.\psi$ or $\exists x.\psi$ of φ.

2. An occurrence of a variable x in a formula φ is *free* if it is not bound.

A variable x *occurs in the scope* of a quantifier in a formula φ if x occurs bound in φ.

Example 3.26.

1. All occurrences of he variable x in the formula $\forall x.\ ((P(x) \to Q(x)) \wedge R(x, f(y, z)))$ are bound; the occurrences of y and z are free.

2. The variable x has a free and a bound occurrence in $\forall x.\ P(x) \wedge Q(x)$.

□

Example 3.27.

1. The function $FV : \Phi \to \mathcal{P}(\mathcal{V})$ defined by

$$\begin{aligned}
FV(\bot) &= \emptyset,\\
FV(\top) &= \emptyset,\\
FV(t_1 = t_2) &= V(t_1) \cup V(t_2),\\
FV(P(t_1, \dots t_n)) &= V(t_1) \cup \cdots \cup V(t_n),\\
FV(\neg\varphi) &= FV(\varphi),\\
FV(\varphi \diamond \psi) &= FV(\varphi) \cup FV(\psi), \quad \text{if } \diamond \in \{\wedge, \vee, \to\},\\
FV(\forall x.\ \varphi) &= FV(\varphi) - \{x\},\\
FV(\exists x.\ \varphi) &= FV(\varphi) - \{x\}
\end{aligned}$$

   computes the set of free variables in a given formula. We tacitly assume that the predicate symbol P has the right arity and $t_1, \dots, t_n$ are terms.

2.

$$\begin{aligned}
&FV(\forall x.\ ((P(x) \to Q(x)) \wedge R(x, f(y, z))))\\
&\quad = FV(((P(x) \to Q(x)) \wedge R(x, f(y, z)))) - \{x\}\\
&\quad = (FV(P(x)) \cup FV(Q(x)) \cup FV(R(x, f(y, z)))) - \{x\}\\
&\quad = (V(x) \cup V(x) \cup V(x) \cup V(f(y, z))) - \{x\}\\
&\quad = (\{x\} \cup \{y\} \cup \{z\}) - \{x\}\\
&\quad = \{y, z\}.
\end{aligned}$$

3.

$$\begin{aligned}
FV(\forall x.\ P(x) \wedge Q(x)) &= FV(\forall x. P(x)) \cup FV(Q(x))\\
&= (FV(P(x)) - \{x\}) \cup FV(Q(x))\\
&= (V(x) - \{x\}) \cup \{x\}\\
&= \emptyset \cup \{x\}\\
&= \{x\}.
\end{aligned}$$

□

3.4.3 Substitution

Another important notion of predicate logic is substitution. Intuitively this means the replacement of all variables in a given formula by a term. A substitution is a function of type $\Phi \to \Phi$ that is completely determined by a "smaller" function of type $\mathcal{V} \to \mathcal{T}$. We write

$$\varphi[t/x]$$

for the result of substituting the term t for each free occurrence of the variable x in φ.

Definition 3.28 (Substitution for Terms)**.** If $s, t \in \mathcal{T}$, then $s[t/x]$ is defined by

$$y[t/x] = \begin{cases} y & \text{if } x \neq y, \\ t & \text{if } x = y, \end{cases} \quad \text{if } s = y \text{ is a variable.}$$

$$c[t/x] = c, \quad \text{if } s = c \text{ is a constant.}$$

$$f(t_1, \ldots, t_n)[t/x] = f(t_1[t/x], \ldots t_n[t/x]), \quad \text{if } s = f(t_1, \ldots, t_n).$$

Definition 3.29 (Substitution for Formulas)**.** If $t \in \mathcal{T}$ and $\varphi \in \Phi$, then $\varphi[t/x]$ is defined by

$$P(t_1, \ldots, t_n)[t/x] = P(t_1[t/x], \ldots, t_n[t/x]), \quad \text{if } \varphi = P(t_1[t/x], \ldots, t_n[t/x]).$$

$$(s_1 = s_2)[t/x] = (s_1[t/x] = s_2[t/x]), \quad \text{if } \varphi = (s_1 = s_2),$$

$$(\neg\psi)[t/x] = \neg(\psi[t/x]), \quad \text{if } \varphi = \neg\psi.$$

$$(\psi_1 \diamond \psi_2)[t/x] = \psi_1[t/x] \diamond \psi_2[t/x], \quad \text{if } \varphi = \psi_1 \diamond \psi_2 \text{ for } \diamond \in \{\wedge, \vee, \to\}.$$

$$(\forall y.\ \psi)[t/x] = \begin{cases} \forall y.\ (\psi[t/x]) & \text{if } x \neq y, \\ \forall y. \psi & \text{if } x = y, \end{cases} \quad \text{if } \varphi = \forall y.\ \psi.$$

$$(\exists y.\ \psi)[t/x] = \begin{cases} \exists y.\ (\psi[t/x]) & \text{if } x \neq y, \\ \exists y. \psi & \text{if } x = y, \end{cases} \quad \text{if } \varphi = \exists y.\ \psi.$$

Example 3.30.

1. $P(x)[a/x] = P(a)$,
2. $P(x, y)[f(y, y)/x] = P(f(y, y), y)$,
3. $(\forall x.\ P(x) \wedge Q(x))[c/x] = \forall x.\ P(x) \wedge Q(c)$,
4. $P(x)[f(y)/x][a/y] = P(f(y))[a/y] = P(f(a))$.

□

A problem that arises with substitutions is *variable capture*: in

$$(\forall x.\ P(x, y))[f(x)/y] = \forall x.\ P(x, f(x)),$$

for instance, the second argument of P is not in the scope of $\forall x$ before the substitution, yet it is afterwards.

Intuitively, we call a term t free for a variable x in a formula φ, if the variables in t do not become bound after substituting t for x in φ. In other words, no free occurrence of x in φ is in the scope of a quantifier $\forall y$ or $\exists y$ for any variable y that occurs in t.

Definition 3.31. A term t is *free for* a variable x *in* a formula φ if either

- φ is atomic,

- $\varphi = \neg\psi$ and t is free for x in ψ,
- $\varphi = \psi_1 \diamond \psi_2$ and t is free for x in ψ_1 and ψ_2,
- $\varphi = \forall y.\ \psi$ and if $x \in FV(\varphi)$, then $y \notin FV(t)$ and t is free for x in ψ,
- $\varphi = \exists y.\ \psi$ and if $x \in FV(\varphi)$, then $y \notin FV(t)$ and t is free for x in ψ.

Example 3.32.

1. y is free for x in $\exists z.\ P(x,z)$.
2. $f(x,y)$ is not free for x in $\exists y.P(x,z)$.
3. v is free for x in $P(x,z) \to \forall x.\ Q(x,y)$.

□

Lemma 3.33. *The term t is free for x in φ if and only if the variables of t in $\varphi[t/x]$ are not bound by a quantifier.*

Proof. Induction on φ. □

We will only apply a substitution of t for x in a formula φ if t is free for x in φ. This can be achieved by renaming.

3.5 Deductive System Revisited

For modelling parameters in natural deduction, we extend the definition of formulas as

$$\mathcal{T} ::= x \mid a \mid f(\mathcal{T}, \ldots, \mathcal{T}),$$

where a is taken from a countably infinite set of parameters. Then

$$\frac{\varphi[a/x]}{\forall x.\ \varphi}\,\forall I \qquad \frac{\forall x.\ \varphi}{\varphi[t/x]}\,\forall E$$

where t is any term which is free for x and φ and a a fresh parameter we know nothing about. Similarly,

$$\frac{[\varphi[t/x]]}{\exists x.\ \varphi}\,\exists I \qquad \frac{\exists x.\ \varphi \qquad \begin{array}{c}[\varphi[a/x]]\\ \vdots \\ \psi\end{array}}{\psi}\,\exists E$$

where t is any term that is free for x in φ and a a fresh parameter we know nothing about.

Finally, in the elimination rule for equality

$$\frac{t_1 = t_2 \qquad \varphi[t_1/x]}{\varphi[t_2/x]} = E$$

t_1 and t_2 must be free for x in φ.

The restriction of terms being free for variables allows us to substitute non-ground terms in deductions. This was not possible with the more naive approach outlined in Section 3.2.

=== END OF WEEK SIX (EXPECTED) ===================

3.6 Semantics

Assigning meanings to formulas of predicate logic is more complicated than for propositional logic, but follows by and large the same recursive approach. It is also more interesting from a modelling point of view, because it tells us how terms and formulas are interpreted or evaluated in concrete structures, such as the natural numbers, boolean algebras, or real-world models. Formal semantics for programming languages can be obtained very much along the same line.

3.6.1 Structures

Remember that a boolean algebra is a (non-empty) set equipped with two binary functions or operations of *join* $\sqcup$ and *meet* $\sqcap$, a unary function or operation of *complementation* $-$, and two constants 0 and 1. In addition, there are some laws that these operations satisfy. In the context of predicate logic, we generalise this further from functions to predicates or relations.

Remember that an *n-ary relation* on a set A is just a subset of $A^n = A \times \cdots \times A$, where the cartesian product $\times$ is taken $n-1$ times. An element of an n-ary relation R is thus an n-tuple $(a_1, \dots, a_n) \in R$ for which $a_1, \dots, a_n \in A$.

Definition 3.34 (Structure)**.** A Σ-*structure* is a pair $\mathfrak{A} = (A, (-)^{\mathfrak{A}})$ of a non-empty set A (the *carrier set* of $\mathfrak{A}$) and a function $(-)^{\mathfrak{A}}$ that associates with each n-ary predicate symbol $R \in \Sigma$ an n-ary relation $R^{\mathfrak{A}} \subseteq A^n$ and with each n-ary function symbol $f \in \Sigma$ an n-ary function $f^{\mathfrak{A}} : A^n \to A$.

Intuitively, $(-)^{\mathfrak{A}}$ is a look-up table that associates function symbols with functions and predicate symbols with relations in a one-to-one fashion.

Functions of arity 0 are called *constants*. We write

$$(A, R_1^{\mathfrak{A}}, \dots, R_m^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \dots, f_n^{\mathfrak{A}})$$

for a structure associated with the signature $(R_1, \dots, R_m, f_1, \dots, f_n)$.

Structures that are solely formed of functions or operations over the carrier set are called *algebraic structures*, those that consist solely of relations are called *relational structures*.

Example 3.35.

1. (Arithmetic). The signature of arithmetic is $\Sigma_A = \{+, \cdot, 0, 1\}$, that is ordered arithmetic is $\Sigma_{OA} = \{+, \cdot, 0, 1, \leq\}$. We assume that the symbols in these signatures are interpreted in the standard way.

   (a) The structure $\mathfrak{N} = (\mathbb{N}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}})$ describes the standard arithmetic of natural numbers. It is a Σ_A-structure. $\mathfrak{N}_{\leq} = (\mathbb{N}, +, \cdot, 0, 1, \leq)$, which describes the ordered arithmetic of $\mathbb{N}$ (we have dropped the superscripts for the sake of readability. It is a Σ_{OA}-structure. The usual laws of arithmetic hold in $\mathfrak{N}$ and $\mathfrak{N}_{\leq}$, for instance, $m + n = n + m$, $m \cdot (n + p) = m \cdot n + m \cdot p$ or $0 \leq n$.

   (b) The ring of integers $\mathfrak{Z} = (\mathbb{Z}, +, \cdot, 0, 1)$ is another Σ_A-structure.

   (c) $\mathfrak{N}_\infty = (\mathbb{N} \cup \{\infty\}, +, \cdot, 0, 1)$ is a Σ_A-structure, in which $n + \infty = \infty + n = n \cdot \infty = \infty \cdot n = \infty$.

2. (Boolean algebras). Their signature is $\Sigma_{BA} = (\sqcap, \sqcup, -, 0, 1)$.

   (a) For every set A, the boolean algebra $\mathfrak{B}_A = (\mathcal{P}(A), \cap, \cup, -, \emptyset, A)$ is a Σ_{BA}-algebra, the boolean algebra over A. Here, $\sqcap$ is interpreted as $\cap$, $\sqcup$ as $\cup$, $-$ as $-$, 0 as $\emptyset$ and 1 as A. Boolean algbras satisfy many laws, for instance $X \cup (X \cap Y) = X$ or $X \cap -X = \emptyset$.

   (b) the booleans $\mathbb{B}$ form the boolean algebra $\mathfrak{B}_{\mathbb{B}} = (\mathbb{B}, \min, \max, 1 - (-), 0, 1)$, hence a Σ_{BA}-algebra.

3. (Graphs). Their signature is $\Sigma_G = \{E\}$ for a binary predicate symbol E. Every Σ_G-structure is a *directed graph.* An *undirected graph* is a Σ_G-structure $\mathfrak{G} = (V, E^{\mathfrak{G}})$ with vertex set V and binary relation $E^{\mathfrak{G}} \subseteq V \times V$ such that $(v, v) \notin E$ for all $v \in V$ and $(v, w) \in E$ implies $(w, v) \in E$ forall $v, w \in V$. The first conditions rules out loops; the second one says that the edge relation is symmetric.

4. (Orderings). The signature of orderings is $\Sigma_O = \{\leq\}$, where $\leq$ is a binary predicate symbol. A Σ_O-structure is any pair $(A, \leq)$ of a non-empty set A equipped with a binary relation $\leq$ (overloading notation). Concrete examples are $(\mathbb{N}, \leq)$ with $m \leq n$ if and only if $m + p = n$ for some $p \in \mathbb{N}$ or $(\mathcal{P}(A), \subseteq)$ for some set A, where $\subseteq$ is the subset relation. Every *partial order*relation satisfies three properties: $x \leq x$ (reflexivity) , $x \leq y$ and $y \leq z$ imply $x = y$ (antisymmetry), and $x \leq y$ and $y \leq z$ imply $x \leq z$ (transitivity), for all $x, y, z \in A$. An ordering is *strict*, if it is an *irreflexive* ($x \not\leq x$ holds for all $x \in A$) transitive relation. A partial order is *linear* or *total* if $x \leq y$ or $y \leq x$ holds for all $x, y \in A$.

Example 3.36.

1. (Labelled Transition Systems). The signature for a labelled transition system is $\Sigma_{LTS} = (\{P_e \mid e \in E\}, \rightarrow)$, where the P_e are unary predicate symbols and $\rightarrow$ is a binary predicate symbol. A *labelled transition system* (*LTS*) is a Σ_{LTS}-algebra $(S, , \{P_e \mid e \in E\}, \rightarrow)$ (overloading notation) such that S is a (finite) set of *states*, a set of

unary predicates $P_e \subseteq S$, which indicate that property e holds in a given state, and a *transition* relation $\rightarrow \subseteq A \times A$, which models a transition of a system from one state into another.

Labelled transition systems can model the behaviour of programs, protocols and control systems, in particular parallel and concurrent ones. They are widely used in software verification, see also Chapter 5. In the context of modal logics, labelled transition systems are related to *Kripke structures*.

2. (Automata). The signature for nondeterministic finite automata is $\Sigma_{NFA} = \{(\{\Delta_\sigma \mid \sigma \in \Sigma\}, S, F\}$, where Σ is a finite set, the *alphabet of the automaton*. A *nondeterministic finite automaton (NFA)* is a Σ_{NFA}-algebra $(Q, \{\Delta_\sigma \mid \sigma \in \Sigma\}, S, F)$ (overloading notation), where Q is a finite set of states, $\Delta_\sigma \subseteq Q \times Q$ is a *transition relation* between states—one for each letter of the alphabet, $S \subseteq Q$ is a unary predicate which indicates an *initial state* and $F \subseteq Q$ a unary predicate which indicates an *accept state*.

The semantics of predicate logic associates structures with languages. The way outlined above is to associate a structure with a language and interpret the syntactic elements of a language within a structure.

From a mathematical and a modelling point of view, the opposite way is very natural as well, namely to associate a language with a structure or model on cares about. In this context one often speaks about a language *for* a structure. We write $\mathcal{L}_{\mathfrak{A}}$ for the language of a structure $\mathfrak{A}$.

Application often require more refined kinds of structures with several carrier sets. An NFA, for instance, is generally modelled as a tuple $(Q, \Sigma, \Delta, S, F)$ with one single transition relation of type $\Delta \subseteq Q \times \Sigma \times Q$. A probability space is a triple (Ω, F, P), where Ω is a set (the sample space), $F \subseteq \mathcal{P}(\Omega)$ the σ-algebra over Q (which has nothing to do with our Σ-structures), and $P : F \to [0, 1]$ a probability measure. Both examples thus require functions or relations between different sets. This can be achieved in the context of *multi-sorted* structures and logics, which is beyond the scope of these lectures.

3.6.2 Models

We now explain how formulas of predicate logics are interpreted in structures, starting with a simple example.

Example 3.37. Consider again the signature $\Sigma_{BA} = (\sqcap, \sqcup, -, 0, 1)$ of boolean algebras and an associated Σ_{BA}-algebra $\mathfrak{B} = (B, \sqcup^{\mathfrak{B}}, \sqcap^{\mathfrak{B}}, -^{\mathfrak{B}}, 0^{\mathfrak{B}}, 1^{\mathfrak{B}})$. That is, there is a one-to one correspondence $(-)^{\mathfrak{B}}$ between the function symbols of boolean algebras and the functions on the concrete boolean algebra $\mathfrak{B}$. In the boolean algebra $\mathfrak{B}_A$ over the set A, for instance, $\sqcap$ is interpreted by $\cap$, $\sqcup$ by $\cup$ and so on. For the sake of simplicity we disregard free variables. First, we need to explain how Σ_{BA}-terms are interpreted in $\mathfrak{B}$. Then we need to determine the truth values of Σ_{BA} formulas. We write $[\![-]\!]$ for the function that inteprets terms and formulas.

- We assume that our language contains constant symbols $c_0, c_1, \ldots$ that are not mentioned in the signature. Their interpretation is straightforward. We interpret them as elements $c_0^{\mathfrak{B}}, c_1^{\mathfrak{B}}, \ldots$ of B, that is, $[\![c_i]\!] = c_i^{\mathfrak{B}}$. In $\mathfrak{B}_A$, for instance, $[\![c_i]\!] = X_i$ for a given $X_i \subseteq A$; $[\![0]\!] = \emptyset$ and $[\![1]\!] = A$.

  The interpretation of a composite term $t_1 \sqcap t_2$ proceeds by recursion, like that of the propositional connectives in Section 2.8.2: $[\![t_1 \sqcap t_2]\!] = [\![t_!]\!] \sqcap^{\mathfrak{B}} [\![t_1]\!]$, and likewise for the other operations. In $\mathfrak{B}_A$, if $[\![t_1]\!] = X$ and $[\![t_2]\!] = Y$, then $[\![t_1 \sqcap t_2]\!] = X \cap Y$ and $[\![t_1 \sqcup -t_1]\!] = X_1 \cup -X = A$.

- An atomic formula $t_1 = t_2$ is interpreted as true if the interpretations of t_1 and t_2 yield the same value, and false otherwise; $[\![t_1 = t_2]\!] = 1$ if and only if $[\![t_1]\!] =]\!]t_2]\!]$. Composite formulas arising from propositional connectives are interpreted as in propositional logic (Definition 2.33). A formula like $\forall x \forall y.\ x \sqcup y = y \sqcup x$ is interpreted as true if and only if $a \sqcup b = b \sqcup a$ holds for all elements $a, b \in A$. Hence the interpretation $[\![(x \sqcup y = y \sqcup x)[c_1/x][c_2/y]]\!] = 1$ for all $c_1^{\mathfrak{B}}, c_2^{\mathfrak{B}} \in B$. Similarly, a formula $\exists x.\ c \sqcap x = 0$ is interpreted as true if and only if $[\![(c \sqcap x = 0)[d/x]]\!] = 1$ for some $d^{\mathfrak{B}} \in B$, that is, if we can find a witness d that makes the formula $c^{\mathfrak{B}} \sqcap^{\mathfrak{B}} d^{\mathfrak{B}}$ true in $\mathfrak{B}$. This witness is of course the interpretation of $-c$ in $\mathfrak{B}$.

□

This approach generalises to arbitrary languages and structures, and we formalise it further. For the remainder of this section we fix a signature $\Sigma = (R_1, \ldots, R_m, f_1, \ldots, f_n)$ and a Σ-structure

$$\mathfrak{A} = (S, R_1^{\mathfrak{A}}, \dot{R}_m^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \ldots, f_n^{\mathfrak{A}})$$

Definition 3.38 (Assignment)**.** An *assignment* (or *environment*) is a function $v : \mathcal{V} \to S$ that associates variables with constants $c^{\mathfrak{A}} \in S$.

Definition 3.39 (Interpretation of Terms)**.** Let $v : \mathcal{V} \to A$ be an assignment. An *interpretation* of terms is a function $[\![-]\!]_v^{\mathfrak{A}} : \mathcal{T} \to S$ defined by

$$\begin{aligned} [\![x]\!]_v^{\mathfrak{A}} &= v(x), && \text{if } v \in \mathcal{V}, \\ [\![c]\!]_v^{\mathfrak{A}} &= c^{\mathfrak{A}}, && \text{if } c \in \mathcal{F} \text{ is a constant}, \\ [\![f(t_1, \ldots, t_n)]\!]_v^{\mathfrak{A}} &= f^{\mathfrak{A}}([\![t_1]\!]_v^{\mathfrak{A}}, \ldots, [\![t_n]\!]_v^{\mathfrak{A}}). \end{aligned}$$

Definition 3.40 (Interpretation of Formulas)**.** Let $v : \mathcal{V} \to A$ be an assignment. We extend

$[\![-]\!]^{\mathfrak{A}}_v$ to an *interpretation* $[\![-]\!]^{\mathfrak{A}}_v : \Phi \to \mathbb{B}$ defined by

$$\begin{aligned}
[\![P(t_1,\dots,t_n)]\!]^{\mathfrak{A}}_v &= \begin{cases} 1, & \text{if } ([\![t_1]\!]^{\mathfrak{A}}_v,\dots,[\![t_n]\!]^{\mathfrak{A}}_v \in P^{\mathfrak{A}}, \\ 0, & \text{otherwise,} \end{cases} \\
[\![\neg\varphi]\!]^{\mathfrak{A}}_v &= 1 - [\![\varphi]\!]^{\mathfrak{A}}_v, \\
[\![\varphi \wedge \psi]\!]^{\mathfrak{A}}_v &= \min([\![\varphi]\!]^{\mathfrak{A}}_v, [\![\psi]\!]^{\mathfrak{A}}_v), \\
[\![\varphi \vee \psi]\!]^{\mathfrak{A}}_v &= \max([\![\varphi]\!]^{\mathfrak{A}}_v, [\![\psi]\!]^{\mathfrak{A}}_v), \\
[\![\varphi \to \psi]\!]^{\mathfrak{A}}_v &= \max(1 - [\![\varphi]\!]^{\mathfrak{A}}_v, [\![\psi]\!]^{\mathfrak{A}}_v), \\
[\![\forall x.\ \varphi]\!]^{\mathfrak{A}}_v &= \min_{c^{\mathfrak{A}} \in S}([\![\varphi]\!]^{\mathfrak{A}}_{v[c^{\mathfrak{A}}/x]}), \\
[\![\exists x.\ \varphi]\!]^{\mathfrak{A}}_v &= \max_{c^{\mathfrak{A}} \in S}([\![\varphi]\!]^{\mathfrak{A}}_{v[c^{\mathfrak{A}}/x]}),
\end{aligned}$$

where $v[a/x]$ indicates that the assignment function v has been updated in x by the value a, that is, for all $a \in S$,

$$v[a/x](y) = \begin{cases} v(y), & \text{if } x \neq y, \\ a, & \text{otherwise.} \end{cases}$$

The interpretation of $\forall x.\ \varphi$ and $\exists x.\ \varphi$ certainly deserves an explanation. By definition, $[\![\forall x.\ \varphi]\!]^{\mathfrak{A}}_v = 1$ if and only if $[\![\varphi]\!]^{\mathfrak{A}}_{v[c^{\mathfrak{A}}/x}$ holds for all $c^{\mathfrak{A}} \in S$. Moreover, $[\![\exists x.\ \varphi]\!]^{\mathfrak{A}}_v = 1$ if and only if $[\![\varphi]\!]^{\mathfrak{A}}_{v[c^{\mathfrak{A}}/x}$ holds for some $c^{\mathfrak{A}} \in S$. Note that we overload the notation for function update and substitution, though these are completely different concepts. The notion of function update is more akin to the update of a program store, when performing an assignment command.

Definition 3.41 (Model Relation)**.**

1. A *model*of a formula φ is a pair of a structure $\mathfrak{A}$ and an assignment v such that $[\![\varphi]\!]^{\mathfrak{A}}_v = 1$. In this case we write
$$\mathfrak{A} \models_v \varphi.$$
2. A pair $(\mathfrak{A}, v)$ is a model of a set $\Gamma \subseteq \Phi$ of formulas if $\mathfrak{A} \models_v \varphi$ for all $\varphi \in \Gamma$. In this case we write
$$\mathfrak{A} \models_v \Gamma.$$

Interpretations of sentences do not depend on assignments in the sense that if an interpretation of a sentence is true, then it is true for any assignment. Hence if φ is a sentence, then we simply write $\mathfrak{A} \models \varphi$. A model of a sentence is thus simply a structure in which the sentence holds.

Definition 3.42 (Model Class)**.** The *model class* of a set $\Gamma \subseteq \Phi$ of sentences consists of all structures $\mathfrak{A}$ such that $\mathfrak{A} \models \Gamma$.

Example 3.43. The model class of the sentences

- $\forall x.\ x \leq x$ (reflexivity),

- $\forall x \forall y.\ x \leq y \wedge y \leq x \rightarrow x = y$ (antisymmetry),
- $\forall x \forall y \forall z.\ x \leq y \wedge y \leq z \rightarrow x \leq y$ (transitivity)

is the class of all partial orders.

The set of sentences that characterises a particular class of structures is called a set of *axioms* for that structure. The reflexivity, antisymmetry and transitivity axioms thus axiomatise the class of partial orders; every partial order must satisfy these axioms. Many mathematical structures can be axiomatised within first-order logic, often with a finite set of equational axioms.

Definition 3.44.

1. A sentence $\varphi \in \Phi$ is a *tautology* if $\mathfrak{A} \models \varphi$ for all structures $\mathfrak{A}$. We write $\models \varphi$ if φ is a tautology.
2. A set $\Gamma \subseteq \Phi$ of sentences *(semantically) entails* a sentence $\varphi \in \Phi$, written $\Gamma \models \varphi$ if every model of Γ is a model of φ, that is, $\mathfrak{A} \models \varphi$ implies $\mathfrak{A} \models \varphi$ for every structure $\mathfrak{A}$.

Definition 3.45 (Satisfiability)**.**

1. A sentence $\varphi \in \Phi$ is *satisfiable* if $\mathfrak{A} \models \varphi$ for some structure $\mathfrak{A}$.
2. φ is *unsatisfiable* (a *contradiction*) if it is not satisfiable.

The extension of these notions to sets of sentences is straightforward.

Lemma 3.46. *If we consider sentences only, then*

1. $\mathfrak{A} \models \neg\varphi$ *if and only if* $\mathfrak{A} \not\models \varphi$,
2. $\mathfrak{A} \models \varphi \wedge \psi$ *if and only if* $\mathfrak{A} \models \varphi$ *and* $\mathfrak{A} \models \psi$,
3. $\mathfrak{A} \models \varphi \vee \psi$ *if and only if* $\mathfrak{A} \models \varphi$ *or* $\mathfrak{A} \models \psi$,
4. $\mathfrak{A} \models \varphi \rightarrow \psi$ *if and only if* $\mathfrak{A} \models \psi$ *whenever* $\mathfrak{A} \models \varphi$,
5. $\mathfrak{A} \models \forall x.\ \varphi$ *if and only if* $\mathfrak{A} \models \varphi[c/x]$ *for all* $c^{\mathfrak{A}} \in A$,
6. $\mathfrak{A} \models \exists x.\ \varphi$ *if and only if* $\mathfrak{A} \models \varphi[c/x]$ *for some* $c^{\mathfrak{A}} \in A$.

Note that in the above formulas, $\varphi[c/x]$ is a substitution.
The following example is taken from Huth and Ryan's book.

Example 3.47. The sentence

None of Alma's lovers' lovers love her.

translates to the sentence

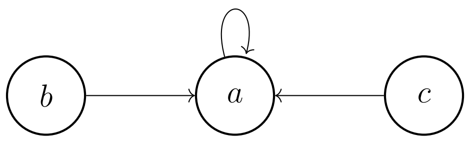
$$\varphi = \forall x \forall y.\ (L(x,a) \wedge L(y,x) \rightarrow \neg L(y,a))$$

over the signature $\Sigma = \{L, a\}$, where L is a binary predicate symbol and a a constant symbol.

1. Consider the Σ-structure $\mathfrak{A} = (A, L^{\mathfrak{A}}, a^{\mathfrak{a}})$ with

   $$A = \{b, c\}, \qquad \text{and} \qquad R^{\mathfrak{A}} = \{(a^{\mathfrak{A}}, a^{\mathfrak{A}}), (b, a^{\mathfrak{A}}), (c, a^{\mathfrak{A}})\}.$$

   We henceforth drop the superscript in $a^{\mathfrak{A}}$. $\mathfrak{A}$ can be visualised by the following graph.

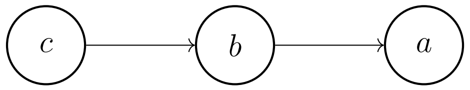
   

   We ask, whether $\mathfrak{A}$ is a model of φ, that is, whether $\mathfrak{A} \models \varphi$ holds. The answer is "no". To refute φ we must find an assignment $x \mapsto x_0$ and $y \mapsto y_0$ for which $K(x_0, a)$ and $L(y_0, x_0)$ are true, but $\neg L(y_0, a)$ is false (hence $L(y_0, a)$ must be true). It is easy to see that $x \mapsto a$ and $y \mapsto b$ does the job: $L(a,a)$ and $L(b,a)$ both hold. Hence $\mathfrak{A} \not\models \varphi$.

2. Now consider the Σ-structure $\mathfrak{B} = (B, L^{\mathfrak{B}}, a^{\mathfrak{B}}\}$ with

   $$B = \{b, ac\} \qquad \text{and} \qquad R^{\mathfrak{B}} = \{(b, a^{\mathfrak{B}}), (c, b)\},$$

   that is,

   In this case, the only assignment that makes $L(x,a)$ and $L(y,x)$ true is $x \mapsto b$ and $y \mapsto c$, yet then $L(c,a)$ is certainly false as there is no arrow from c to a. Hence $\mathfrak{B} \models \varphi$.

□

3.7 Soundness, Completeness, Compactness

Like for propositional logic, we write $\Gamma \vdash \varphi$ if the formula φ is derivable from the set Γ of formulas by natural deduction. It is once again the case that $\Gamma \vdash \varphi$ implies $\Gamma_0 \vdash \varphi$ for some finte $\Gamma_0 \subseteq \Gamma$.

Theorem 3.48 (Soundness and Completeness). *Let $\varphi \in \Phi$ be a sentence and $\Gamma \subseteq \Phi$ a set of sentences. Then*

$$\Gamma \vdash \varphi \Leftrightarrow \Gamma \models \varphi.$$

The proof of soundness is similar to that of propositional logic. It is left as an exercise. The completeness proof is complicated and beyond the scope of these lectures. It follows by and large the structure of the completeness proof of propositional logic, and can be found in textbooks on mathematical logic.

Theorem 3.49 (Compactness)**.** *For every set of sentences $\Gamma \subseteq \Phi$ and for every $\varphi \in \Phi$, $\Gamma \models \varphi$ if and only if there is a finite $\Gamma_0 \subseteq \Gamma$ with $\Gamma_0 \subseteq \varphi$.*

Proof. Same as for propositional logic (Theorem 2.56). □

It follows that a set $\Gamma \subseteq \Phi$ of sentences is satisfiable if and only if every finite subset Γ_0 of Γ is satisfiable.

The compactness theorem allows us to transfer results about finite structures to infinite ones.

Lemma 3.50. *If Γ has models of arbitrary finite size, then it has an infinite model.*

Proof. Let $\Delta = \Gamma \cup \{\varphi_{\geq n} \mid n \geq 2\}$, where the formula

$$\varphi_{\geq n} = \exists x_1 \dots \exists x_n. \bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)$$

expresses the fact that there are at least n different elements. Hence every model of Δ is a model of Γ with infinitely many elements. By compactness it suffices to show that every finite subset Δ_0 of Δ has a model. But each such set satisfies $\Delta_0 \subset \Gamma \cup \{\varphi_{\geq n} \mid 2 \leq n \leq n_0\}$ for some number $n_0 \in \mathbb{N}$.

By the assumption, Γ has a model with at least n_o elements. Hence this model is also a model of $\Gamma \cup \{\varphi_{\geq n} \mid 2 \leq n \leq n_0\}$, and consequently of Δ_0. □

As a computing application of compactness we show that reachability in directed graphs cannot be expressed in predicate logic with a single binary relation symbol E—corresponding to the edge relation of a graph.

Lemma 3.51. *Let $\Sigma_G = \{E\}$ be the signature for directed graphs from Example 3.12. There is no formula $\varphi(x, y)$ with free variables x and y in this language which is true in all Σ_G-structures $\mathfrak{G}$, as introduced in Example 3.35, if and only if there exists a finite path from the node assigned to x to the node assigned to y in $\mathfrak{G}$.*

Proof. Suppos that $\varphi(x, y)$ does express reachability in graphs and consider the set

$$\Gamma = \varphi(x, x) \wedge \{\neg E^n(x, y) | n \in \mathbb{N}\},$$

where $E^n(x, y)$ says that there exists a path of length n from any node we may assign to x to any node we may assign to y in a graph. It can be defined recursively by $E^0(x, y) = (x = y)$ and $E^{n+1} = \exists z.\ E^n(x, z) \wedge E(z, y)$. Clearly, Γ is unsatisfiable. Yet every finite subset of Γ is satisfiable: there must be a maximal m for which $\neg E^m(x, y)$ is in each such subset, hence every graph with a path of length $> m$ yields a model. But then Γ has a model, too, by compactness. This is a contradiction. □

Theorem 3.52. *The satisfiability (and hence the validity) problem for predicate logic is undecidable.*

3.8 Modelling Examples

3.8.1 Sudoku Revisited

This section describes a formalisation of Sudoku in predicate logic. It is taken from an input file for the Mace4 counterexample generator to the Prover9 automated theorem prover developed by William McCune. The original code can be found online at

https://www.cs.unm.edu/~mccune/sudoku/

Instructions for downloading and using Prover9 and Mace4 can be found there as well.

The specification does not model the 9×9 grid directly. Instead, Mace4 is called on a domain of size 9. Let $\lambda(i, j)$ denote the number that labels the cell (i, j). For instance, $\lambda(3, 7) = 2$ means that cell in the third row and the seventh column is labelled with 2.

It is easy to specify that each colour appears at most once in every row and every column with predicate logic:

$$\forall i \forall j_1 \forall j_2.\ (\lambda(i, j_1) = \lambda(i, j_2) \to j_1 = j_2), \qquad \forall i_1 \forall i_2 \forall j.\ (\lambda(i_1, j) = \lambda(i_2, j) \to i_1 = i_2).$$

To specify the corresponding rule for regions, one must express that two cells in the same region that have the same label must have the same coordinates. Hence it remains to express when two cells are in the same region. This is the case if they are in the same row interval and in the same column interval. These are [0..2], [3..5] and [6..8] in both dimensions. Let $i \sim j$ denote that i and j are in the same interval. Obviously, $\sim$ is an equivalence relation, that is, reflexive, symmetric and transitive:

$$i \sim i, \qquad i \sim j \to j \sim i, \qquad i \sim j \wedge j \sim k \to i \sim k.$$

It is then specified completely on the 9×9 by the constraints $0 \sim 1$, $1 \sim 2$, $3 \sim 4$, $4 \sim 5$, $6 \sim 7$, $7 \sim 8$, $0 \sim 3$, $3 \sim 6$ and $0 \sim 6$. The constraint that number appears at most once in each regions can now be expressed as

$$\forall i_1 \forall i_2 \forall j_1 \forall j_2.\ (i_1 \sim i_2 \wedge j_1 \sim j_2 \wedge \lambda(i_1, j_1) \wedge \lambda(i_2, j_2) \to j_1 = i_2 \wedge j_1 = j_2).$$

The conjunction of these "at most rules" is sufficient for solving Sudoku puzzles from a given initial configuration. Using terms forces that there must be a label for each cell of the grid, and hence exactly each number once in each row, column and region. In addition McCune found that adding "at least" rules for rows and columns increased the performance of Mace4:

$$\forall i \forall n \exists j.\ \lambda(i, j) = n, \qquad \forall j \forall n \exists i.\ \lambda(i, j) = n$$

He omitted a similar rule for regions as it was less straightforward to specify and less helpful in practice.

To solve a particular puzzle, it then remains to specify the initial configuration, which is a list of equations $\lambda(i, j) = k$ for fixed values $i, j, k \in [0..8]$ and conjoin them with the other constraints. Such a configuration can be found online in the Mace4 input file.

The logical specification of the Sudoku puzzle is incomplete in that the dimension of the grid and the range of numbers has not been specified within predicate logic. A complete specification could have been obtained by declaring that each variable is either equal to 1 or to 2 and so on within each formula. Yet this would have lead to more cluttered specifications.

The model of Sudoku in predicate logic is thus quite different from the propositional one in Section 2.10.1, and probably simpler. Using the quantifier $\exists!$, which denotes unique existence, and which can be defined as $(\exists! x.\varphi(x) = \exists x.(\ \varphi(x) \wedge \forall y.(\varphi(y) \rightarrow x = y))$, one could even write

$$\forall i \forall n \exists! j.\ \lambda(i,j) = n, \qquad \forall j \forall n \exists! i.\ \lambda(i,j) = n$$

to capture the "at most" and "at least" conditions for rows and columns more concisely, but this feature is not built into the input language of Mace4.

3.8.2 Graph Colouring Revisited

In predicate logic, the three constraints that describe colorability of a graph in Section 2.10.2 can now be expressed more directly. In a first approach, $V(x)$ denotes that x is a vertex, $E(x,y)$ that there is an edge from x to y and $C(x,y)$ that x had colour y. We can then write

$$\begin{aligned} C_1 &= \forall x.\ (V(x) \rightarrow \exists y.\ C(x,y)),\\ C_2 &= \forall x \forall y_1 \forall y_2.\ (C(x,y_1) \wedge C(x,y_2) \rightarrow y_1 = y_2),\\ C_3 &= \forall x_1 \forall x_1.\ E(x_1,x_2) \rightarrow \neg \exists y.\ C(x_1 y) \wedge C(x_2,y). \end{aligned}$$

As in Section 2.10.2, C_1 states that every vertex of the graph has at least one colour, C_2 that every vertex has at most one colour and C_3 that not adjacent vertices have the same colour.

As in the case of Sudoku, however, the constraints $C_!$ and C_2 become unnecessary if we use the labelling function $\lambda(x)$, which assigns a colour to each vertex x directly, instead of $C(x,y)$. We can then replace C_3 simply by

$$\forall x_! \forall x_2.\ E(x_1,x_3) \rightarrow \lambda(x_1) \neq \lambda(x_2).$$

To verify that a given graph is k-coloured, it seems desirable to make a distinction between vertices and colours at the semantic level. The most appropriate way of formalising this is in a two-sorted setting, which has been outlined at the end of Section 3.6.1. This means that structures with two carrier sets, one for vertices and one for colours, should be used; the labelling function would then map from the first sort to the second one. Details are beyond the scope of these lectures.

Note, however, that like reachability, k-colourability for each $k \geq 2$ is not expressible in predicate logic.

3.8.3 Relational Databases

The following table represents a simple university data base.

Student	Department	Tutor
Alice	Computer Science	von Neumann
Bob	Aerospace	Gagarin
Carlos	Mathematics	Ramanujan
Djamila	Mathematics	Fermat

The order of the rows in the table is irrelevant and it makes sense to store each piece of information in a row only once. The table can thus be modelled as a set of triples

$$\begin{aligned}&\{(\mathit{Alice}, \mathit{Computer\ Science}, \mathit{von\ Neumann}),\\ &\ (\mathit{Bob}, \mathit{Aerospace}, \mathit{Gagarin}),\\ &\ (\mathit{Carlos}, \mathit{Mathematics}, \mathit{Ramanujan}),\\ &\ (\mathit{Djamila}, \mathit{Mathematics}, \mathit{Fermat})\},\end{aligned}$$

hence as a ternary relation over the domain

$$\begin{aligned}D = \{&\mathit{Alice}, \mathit{Computer\ Science}, \mathit{von\ Neumann},\\ &\mathit{Bob}, \mathit{Aerospace}, \mathit{Gagarin},\\ &\mathit{Carlos}, \mathit{Mathematics}, \mathit{Ramanujan}\\ &\mathit{Djamila}, \mathit{Mathematics}, \mathit{Fermat}\}.\end{aligned}$$

of the database. Using one single domain makes the data very unstructured. Nobody would prevent users from adding triples such as

$$(\mathit{Mathematics}, \mathit{Alice}, \mathit{Aerospace}).$$

to the table. In real-world databases this is prevented by using various domains of different sort or type. Yet this is beyond the scope of this simple example.

Databases usually contain many tables and they evolve over time. At each instance of time they can be seen as collections of tables over a carrier set D—or a set of carrier sets—and hence a relational structure $\mathfrak{D} = (D, R_1^{\mathfrak{D}}, \dots, R_n^{\mathfrak{D}})$ over a relational signature $(R_1, \dots, R_n)$. A database can thus be identified with the class of all relational structures of a given signature over a given domain.

In the parlance of data bases, signatures are called *relational schemas*, the corresponding relational structures *relational instances*, and the underlying carrier sets *domains*. Domains can be infinite, but only finitely many elements can occur in a given instance. This *active domain* of a database instance is thus finite by definition.

Querying a database is as important as storing data. Intuitively, a query on a given database instance returns a set of tuples as an answer. We will see examples below. Over finite (active) domains, queries can be effectively evaluated. Predicate logic is an obvious candidate for a query language for relational data bases—and this was known since the beginning of the field. Real-world query languages such as Datalog are fragments of predicate logic that are more domain-specific and computationally superior. Next we explain predicate logic as a query language for relational data bases.

Let $\mathfrak{D} = (D, R_1^{\mathfrak{D}}, \dots, R_n^{\mathfrak{D}})$ be a database instance with (active) domain D. Let φ be a formula of predicate logic. We write

$$Q(\mathfrak{D}, \varphi) = \{v \mid \mathfrak{D} \models_v \varphi\}.$$

for the set of all valuations v for which $\mathfrak{D}$ is a model of φ.

For the above university database, for instance, let $\Sigma = \{\mathit{University}\}$ and let $\mathit{University}^{\mathfrak{D}}$ be the ternary relation described by its table. For the formula

$$\varphi = \mathit{University}(x, \mathit{Mathematics}, y),$$

$Q(\mathfrak{D}, \varphi)$ consists of the two assignments

$$v_1 : x \mapsto \mathit{Carlos}, y \mapsto \mathit{Ramanujan} \qquad \text{and} \qquad v_2 : x \mapsto \mathit{Djamila}, y \mapsto \mathit{Fermat}.$$

We can represent the values v_1 and v_2 as a set of pairs, in this case

$$\{(\mathit{Carlos}, \mathit{Ramanujan}), (\mathit{Djamila}, \mathit{Fermat})\},$$

hence as a binary relation. More generally, therefore, $Q(\mathfrak{D}, \varphi)$ can be represented by a k-ary relation, whenever φ has k free variables. The corresponding function, which maps database instances to k-ary relations, is called a database *query*. Here are some more examples for our university database.

1. We wish to list all students in the mathematics department who have Ramanujan as a tutor. The corresponding query is

$$\mathit{University}(x, \mathit{Mathematics}, \mathit{Ramanujan}).$$

   The answer is the "unary relation" $\{\mathit{Carlos}\}$. For a bigger table in which Ramanujan has more than one tutee, the answer would be a set of student names.

2. We wish to list all departments in which Alice is registered as a student:

$$\exists y.\, \mathit{University}(\mathit{Alice}, x, y).$$

   The quantification over all tutors ensures that these are ignored in the answer. In our example, the answer is $\{\mathit{Computer\ Science}\}$.

3. We wish to list all students who are their own tutors:

$$\exists y.\, \mathit{University}(x, y, x).$$

   Now, the answer is the empty binary relation. In a sports database, by contrast, queries about athletes who are their own coach might lead to non-empty answer sets.

4. We wish to list the entire database:

$$University(x, y, z).$$

In databases with several tables, logical connectives can be used for linking queries. If our university database contains another table which lists the gender of all students, we can ask, for instance, for a list of all female students who study aerospace engineering. This is beyond the scope of these lectures.

Finally, it should be stressed that this example yields a simplistic and even somewhat unrealistic view on databases, though the general relationship between logic and databases is rather strong.

3.9 List of Inference Rules of Natural Deduction

3.9.1 Intuitionistic Rules

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi}\wedge I \qquad\qquad \frac{\varphi \wedge \psi}{\varphi}\wedge E_l \qquad\qquad \frac{\varphi \wedge \psi}{\psi}\wedge E_r$$

$$\frac{\begin{matrix}[\varphi]\\ \vdots \\ \psi\end{matrix}}{\varphi \to \psi}\to I \qquad\qquad \frac{\varphi \to \psi \qquad \varphi}{\psi}\to E$$

$$\frac{}{\top}\top I \qquad\qquad \frac{\bot}{\varphi}\bot E$$

$$\frac{\varphi}{\varphi \vee \psi}\vee I_l \qquad\qquad \frac{\psi}{\varphi \vee \psi}\vee I_r \qquad\qquad \frac{\varphi \vee \psi \qquad \begin{matrix}[\varphi]\\ \vdots \\ \chi\end{matrix} \qquad \begin{matrix}[\psi]\\ \vdots \\ \chi\end{matrix}}{\chi}\vee E$$

$$\frac{\begin{matrix}[\varphi]\\ \vdots \\ \bot\end{matrix}}{\neg\varphi}\neg I \qquad\qquad \frac{\varphi \qquad \neg\varphi}{\bot}\neg E$$

$$\frac{\varphi[a/x]}{\forall x.\ \varphi}\,\forall I \qquad \frac{\forall x.\ \varphi}{\varphi[t/x]}\,\forall E$$

where t is any term and a a fresh parameter we know nothing about.

$$\frac{[\varphi[t/x]]}{\exists x.\ \varphi}\,\exists I \qquad \frac{\exists x.\ \varphi \qquad \begin{matrix}[\varphi[a/x]]\\ \vdots \\ \psi\end{matrix}}{\psi}\,\exists E$$

where t is any term and a a fresh parameter we know nothing about.

3.9.2 Classical Rules

$$\frac{}{\varphi \vee \neg\varphi}\,\text{lem} \qquad \begin{matrix}[\neg\varphi]\\ \vdots \\ \dfrac{\bot}{\varphi}\,\text{pbc}\end{matrix} \qquad \frac{\neg\neg\varphi}{\varphi}\,\neg\neg E$$

3.9.3 Rules for Equality

$$\frac{}{\text{t=t}} = I \qquad \frac{t_1 = t_2 \qquad \varphi[t_1/x]}{\varphi[t_2/x]} = E$$

Chapter 4

Automated Proof Search

In Chapter 2 and 3 we have studied propositional and predicate logic both from a foundational and a formal mathematical point of view. We have introduced natural deduction as a general formalism for mathematical proofs and general reasoning, and we have provided formal syntax and semantics that can be programmed on a computer. Mathematical proofs can therefore be performed on a machine. There are in general two approaches to mechanised logical reasoning. The first one consists in implementing the laws of natural deduction or a similar deductive formalism on a machine. This generally leads to *interactive theorem proving*, where mathematical proofs are typed in step by step by a user, and their correctness is checked step by step by the machine. Interactive theorem proving tools are therefore often called *proof assistants*. The approach is usually automated to some extent, but natural deduction does not seem particularly useful for searching proofs.

If proof automation is not a main goal, then a limitation to predicate logic seems unnatural as well. In fact, most existing proof assistants are based on typed higher-order logics, which allow quantification over functions and predicates. This, in particular, supports proofs by induction, which is beyond the scope of predicate logic.

At the other end of the spectrum are tools that perform automated proof search. These analyse the validity, satisfiability or unsatisfiability of a formula of a formula of propositional or predicate logic fully automatically. From a theoretical point of view, such approaches seem doomed: already the SAT problem of propositional logic is NP-hard and therefore intractable; in predicate logic, validity, satisfiability and unsatisfiability is even undecidable. Yet *SAT-solvers* for propositional logics and *automated theorem provers* for full predicate logic with equality perform surprisingly well in practice. They are therefore widely used for analysing and verifying computing and engineering systems, and have even be able to prove some combinatorially difficult mathematical problems fully automatically.

The main aim of automated proof search is therefore efficiency. Whether the proof search algorithms follow the human way of reasoning, and whether we can make sense of the proof objects they supply (if they do) is not a concern. It is therefore rather unsurprising that the deductive formalisms that have been developed for automated theorem proving for more than 50 years are quite different from natural deduction. Their study is the goal of this chapter.

First, returning to propositional logic, we introduce two of the most important methods for automated proof search, namely propositional Davis-Putnam-Logemann-Loveland algorithm (or DPLL-algorithm) and the propositional resolution method. Then we show how the resolution method can be extended to full predicate logic. The inputs for all methods are formulas that are in a certain normal form. And this is where we start.

4.1 Conjunctive Normal Form

We have seen in Chapter 2 that, in classical propositional logic, only a reduced set of propositional connectives is needed. Here we consider a reduced syntax of the form

$$\Phi ::= p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi,$$

where $p \in P$ for a set P of propositional variables. Using this reduced syntax, we can represent formulas in normal forms similar to the polynomials from arithmetic.

Theorem 4.1 (Conjunctive Normal Form)**.** *Every formula of φ (classical) propositional logic is logically equivalent to a formula in* conjunctive normal form *(*CNF*), that is,*

$$\varphi \equiv \bigwedge_i \bigvee_j L_{ij},$$

where each L_{ij} is either in the set $V(\varphi)$ of propositional variables that occur in φ, or the negation of an element of $V(\varphi)$.

Proof. Exercise. □

The proof of Theorem 4.1 completely algorithmic. It allows us to construct the CNF formula from the input formula by using well known equivalences from propositional logic. Of course, by soundness and completeness of propositional logic we know that

$$\vdash \varphi \leftrightarrow \psi \Leftrightarrow \varphi \equiv \psi.$$

Before describing the CNF algorithm in detail we provide some definitions.

Definition 4.2 (Literals, Clauses, CNF Formulas)**.** Let P be a set of propositional variables.

1. The set of *literals* over P is defined, for $p \in P$, by the grammar

   $$\mathcal{L} ::= p \mid \neg p.$$

   A *positive literal* is a propositional variable; a *negative literal* is the negation of a propositional variable.

2. The set of *clauses* over P is defined by the grammar

   $$\mathcal{D} ::= \bot \mid \mathcal{L} \mid \mathcal{L} \vee \mathcal{D}.$$

3. The set of all *CNF-formulas* over P is defined by the grammar

$$\mathcal{C} ::= \top \mid \mathcal{D} \mid \mathcal{C} \wedge \mathcal{D}.$$

Example 4.3. The formula $(\neg p \vee q \vee r) \wedge (\neg p \vee r) \wedge \neg q$ is in CNF.

It is often convenient to represent clauses as sets of literals and CNF formulas as *clause sets*. To this end, one associates the empty clause, denoted $\square$, with $\bot$, a clause $C_i = \{L_{i1}, \dots L_{in}\}$ with each disjunction $\bigvee_{j=i}^{n} L_{ij}$, the empty clause set $\emptyset$ with $\top$ and a clause set $\mathcal{C} = \{C_1, \dots C_m\}$ with each conjunction $\bigwedge_{i=1}^{m} C_i = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} L_{ij}$. The set notation is preferable for writing down proofs, because multiple occurrences of literals and clauses have been implicitly reduced. We move freely between both notations.

There are two clause sets that do not contain any literal: the empty clause set $\emptyset$, which corresponds to $\top$, and the set $\{\square\}$, which contains the empty clause and corresponds to $\bot$.

Example 4.4. Written as a clause set, the above CNF-formula is $\{\{\neg p, q, r\}, \{\neg p, r\}, \{q\}\}$.

The CNF-algorithm takes a propositional formula from Φ as an input and returns an equivalent formula in CNF. When presented in a declarative way, it performs four steps. Each of them uses equivalences between propositional formulas from Chapter 2, which rewrite or simplify formulas from left to right.

- (Step 1). Eliminate $\leftrightarrow$ and $\rightarrow$ from the input formula:

$$\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi),$$
$$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi.$$

- (Step 2). Push negations towards literals:

$$\begin{aligned} \neg\neg\varphi &\equiv \varphi, \\ \neg(\varphi \wedge \psi) &\equiv \neg\varphi \vee \neg\psi, \\ \neg(\varphi \vee \psi) &\equiv \neg\varphi \wedge \neg\psi, \\ \neg\bot &\equiv \top, \\ \neg\top &\equiv \bot. \end{aligned}$$

- (Step 3). Apply distributivity:

$$\varphi \vee (\psi \wedge \chi) \equiv (\varphi \vee \psi) \wedge (\varphi \vee \chi),$$
$$(\varphi \wedge \psi) \vee \chi \equiv (\varphi \vee \chi) \wedge (\psi \vee \chi).$$

- (Step 4). Simplify the resulting formulas in CNF as follows:
  - Replace every clause that contains complementary literals (e.g. P and $\neg P$) by $\top$.

– Simplify every formula using $\varphi \wedge \top \equiv \varphi$, , $\varphi \wedge \bot \equiv \bot$, $\varphi \vee \top = \top$ and $\varphi \vee \bot = \varphi$.
– Delete every clause C if all its literals occur in another clause C".

It is not hard to see that the CNF algorithm terminates, but a formal proof is somewhat tedious. In particular, the right-hand sides of some of the rules used look more complex than their left-hand sides.

Example 4.5.

1.

$$\begin{aligned} P \vee Q \to Q \vee R &\equiv \neg(P \vee Q) \vee (Q \vee R) && \text{(Step 1)} \\ &\equiv (\neg P \wedge \neg Q) \vee (Q \vee R) && \text{(Step 2)} \\ &\equiv (\neg P \vee Q \vee R) \wedge (\neg Q \vee Q \vee R) && \text{(Step 3)} \\ &\equiv \neg P \vee Q \vee R && \text{(Step 4)} \end{aligned}$$

As a clause set, this yields $\{\{\neg P, Q, R\}\}$.

2.

$$\begin{aligned} ((P \to Q) \to P) \to P &\equiv \neg(\neg(\neg P \vee Q) \vee P) \vee P && \text{(Step 1)} \\ &\equiv (\neg\neg(\neg P \vee Q) \wedge \neg P) \vee P && \text{(Step 2)} \\ &\equiv ((\neg P \vee Q) \wedge \neg P) \vee P && \text{(Step 2)} \\ &\equiv (\neg P \vee Q \vee P) \wedge (\neg P \vee P) && \text{(Step 3)} \\ &\equiv \top && \text{(Step 4)} \end{aligned}$$

As a clause set, this yields $\emptyset$.

□

4.2 SAT Solvers

4.2.1 The DPLL Algorithm

The *Davis-Putnam-Logemann-Loverlace algorithm*, or *DPLL algorithm* takes CNF formulas φ of propositional logic as inputs and solves the SAT problem, that is, it returns "*Yes*" if $[\![\varphi]\!]_v = 1$ holds for some assignment v, and otherwise "*No*". It may return the satisfying assignment, too.

The following equivalences are helpful for explaining the algorithm.

$$\begin{aligned} \varphi \wedge \top &\equiv \varphi, \\ \varphi \wedge (\varphi \vee \psi) &\equiv \varphi, \\ \varphi \wedge (\neg\varphi \vee \psi) &\equiv \varphi \wedge \psi, \\ (\varphi \wedge \psi) \vee (\varphi \wedge \neg\psi) &\equiv \varphi. \end{aligned}$$

The CNF algorithm presented in the previous section can be used as a preprocessor, whenever formulas are not in CNF, but only the first three steps of the CNF algorithm need to be used.

Once again we present a declarative variant of the algorithm, which is really straightforward, and assume that it operates on clause sets. The DPLL algorithm has the following five steps.

- (Step 1). Delete all tautologies $\{P, \neg P, \dots\}$.
- (Step 2) For each unit clause $\{L\}$,
  - delete all clauses that contain L,
  - delete $\neg L$ from all clause.
- (Step 3). Delete all clauses that contain pure literals. (A literal is *pure* if no clause contains its negation.)
- (Step 4). If the empty clause is generated, then the original clause set is not satisfiable, and the algorithm returns "*No.*". If the empty set of clauses is generated; then the original set is satisfiable, and the algorithm returns "*Yes.*" (and perhaps the satisfying assignment).
- (Step 5). Performa a *case split* $\mathcal{C}[\top/L]$ and $\mathcal{C}[\bot/L]$ on some literal L, automatically deleting $\bot$ and $\neg\top$ from every clause, as well as every clause containing $\top$ and $\neg\bot$. Apply (Step 1) to (Step 4) recursively to the subcases.

(Step 1) is an equivalence transformation; it is justified by the first equivalence above. The second and third equivalence justifies that the unit clause transformations (Step 2) are equivalence transformations. In (Step 4), the empty clause (and empty conjunction) corresponds to $\bot$, which is 0 under any assignment. The empty clause set (and empty disjunction) corresponds to $\top$, which is 1 under any assignment. As to (Step 4), deletion of pure literals preserves (un)satisfiability. A clause with a pure literal can always be made true. The split rule (Step 5) is based on the fourth equivalence above.

In the DPLL algorithm, (Step 1) to (Step 4) are usually applied eagerly; (Step 5) is applied lazily. The check whether a literal is pure in (Step 3) can be computationally expensive and therefore not always used by SAT solvers.

Lemma 4.6. *A clause set $\mathcal{C}$ is unsatisfiable if and only if $\mathcal{C}[\top/L]$ and $\mathcal{C}[\bot/L]$ both are.*

Proof. Suppose $\mathcal{C}$ is unsatisfiable and let L occur in $\mathcal{C}$ (otherwise the proof is trivial). By the law of excluded middle, L is either true or false, and $\mathcal{C}$ must be false in both cases. Hence $\mathcal{C}[\top/L]$ and $\mathcal{C}[\bot/L]$ are both unsatisfiable.

Suppose $\mathcal{C}$ is satisfiable and let L occur in $\mathcal{C}$ (otherwise the proof is trivial). By the law of excluded middle, $\mathcal{C}$ at least for one of the assignments corresponding to $\mathcal{C}[\top/L]$ and $\mathcal{C}[\bot/L]$. Hence either $\mathcal{C}[\top/L]$ or $\mathcal{C}[\bot/L]$ must be satisfiable. $\square$

It follows that $\mathcal{C}$ is satisfiable if and only if one of $\mathcal{C}[\top/L]$ and $\mathcal{C}[\bot/L]$ is.

Theorem 4.7. *The DPLL algorithm is sound and complete (and it terminates).*

Proof. The algorithm is sound because all rules preserve at least (un)satisfiability. It is complete because in the worst case one expands the whole truth table for a clause set. It terminates because each step produces either a smaller clause set or leads to clause sets with fewer literals. $\square$

Lemma 4.8. *The formula $P \vee Q \to Q \vee R$ is not valid.*

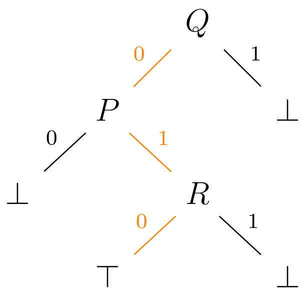
Proof. We use the DPLL algorithm to show that $\neg(P \vee Q \to Q \vee R)$ is satisfiable. The CNF algorithm computes the clause set $\mathcal{C} = \{\{P, Q\}, \{\neg Q\}, \{\neg R\}\}$. The DPLL algorithm reduces it as follows:

$$\begin{aligned} \mathcal{C} &\rightsquigarrow \{\{P\}, \{\neg Q\}\} && \text{unit clause } \{\neg Q\} \\ &\rightsquigarrow \{\{\neg R\}\} && \text{unit clause } \{P\} \\ &\rightsquigarrow \emptyset && \text{unit clause } \{\neg R\} \end{aligned}$$

The algorithm terminates upon the empty clause set $\{\}$ and returns "*Yes.*". This means that the input clause set $\mathcal{C}$ is satisfiable, hence the original clause, which is equivalent to $\neg\mathcal{C}$, is not valid. Note that the last two steps could also have been seen as pure literal steps.

Next we explain how the DPLL algorithm can compute a satisfying assignment for $\mathcal{C}$, hence an assignment for which the original formula is false—a counterexample to validity.

Intuitively, the unit steps performed determine a partial valuation v of the variables in $\mathcal{C}$ for which $[\![\mathcal{C}]\!]_v = 1$. it is first constructed for Q, then extended P and then to R, following the steps in the algorithm. This is illustrated by the following tree.



In the unit clause step with $\{\neg Q\}$, the assignment $Q \mapsto 1$ would have yielded $[\![\mathcal{C}]\!] = 0$, and hence $v : Q \mapsto 0$ is forced. Then $[\![\{\neg Q\}]\!]_v = 1$ and the clause can be deleted from the clause set (conjunction of clauses). Similarly, $[\![Q]\!]_v = 0$ and the literal can be deleted from the clause $\{P, Q\}$ (disjunction of literals). By the same argument, the unit clause step with $\{P\}$ forces $v : P \mapsto 1$ and that with $\{\neg R\}$ forces $v : R \mapsto 0$. The orange path in the tree shows the complete assignment v for which $[\![\mathcal{C}]\!]_v = 1$. $\square$

More generally, we can see unit clause steps as forcing partial assignments. Those with a positive literal P force $P \mapsto 1$, those with a negative literal $\neg P$ force $P \mapsto 0$.

Lemma 4.9. *The following clause set is unsatisfiable:*

$$\mathcal{C} = \{\{\neg Q, R\}, \{\neg R, P\}, \{\neg R, Q\}, \{\neg P, Q, R\}, \{P, Q\}, \{\neg P, \neg Q\}\}.$$

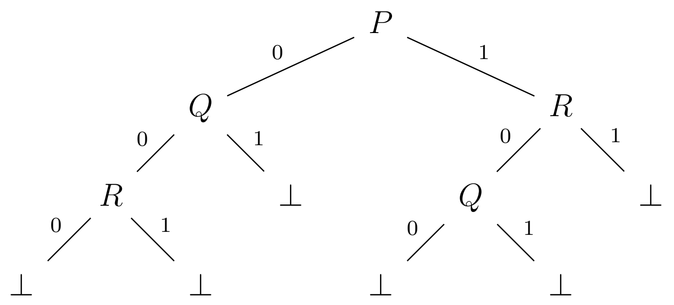
Proof. In this case, none of the rules in (Step 1) - (Step 4) are applicable to $\mathcal{C}$; hence we choose to split on the literal P. In the first case,

$$\begin{aligned} \mathcal{C} &\rightsquigarrow \{\{\neg Q, R\}, \{\neg R, Q\}, \{Q, R\}, \{\neg Q\}\} && \text{case } P \mapsto 1 \\ &\rightsquigarrow \{\{\neg R\}, \{R\}\} && \text{unit clause } \{\neg Q\} \\ &\rightsquigarrow \{\Box\} && \text{unit clause } \{R\} \end{aligned}$$

In the second case,

$$\begin{aligned} \mathcal{C} &\rightsquigarrow \{\{\neg Q, R\{, \{\neg R\}, \{\neg R, Q\}, \{Q\}\} && \text{case } P \mapsto 0 \\ &\rightsquigarrow \{\{\neg Q\}, \{Q\}\} && \text{unit clause } \{\neg R\} \\ &\rightsquigarrow \{\Box\} && \text{unit clause } \{Q\} \end{aligned}$$

Hence in both cases the empty clause $\Box$ has been derived. $\mathcal{C}$ is therefore unsatisfiable.



At the second level of the tree, the assignments $P \mapsto 0$ and $R \mapsto 0$ are forced, and so are the assignments $R \mapsto 1$ and $Q \mapsto 1$ in the third level. Note that all leaves of the tree are labelled with $\bot$, hence $[\![\mathcal{C}]\!]_v = 0$ for all possible assignments v. $\Box$

The branches of the trees in the proofs of Lemma 4.8 and 4.9 correspond to rows of a truth tables. The obvious gain of the DPLL algorithm is that some branches need not be fully expanded. Partial assignments often suffice to make CNF formulas false. Only case splits require full expansions of variables. That is why they should be applied lazily.

The question on which variable to split is crucial for the performance of the algorithm. Finding the best strategy is probably as hard as solving the SAT problem itself. The development of heuristics is crucial for the performance of SAT solvers.

More than half a century after its invention, the DPLL algorithm is still at the core of most modern SAT solvers, and an area of extensive research. More recent extensions include so-called SMT solvers that combine SAT solving with integrated decision procedures for mathematical structures over the natural or the rational numbers or those for data structures such as bit vectors, lists or arrays. These extensions are particularly useful for hardware and software verification.

4.2.2 Propositional Resolution

Resolution-based theorem proving is almost as old as the DPLL algorithm. For propositional logic, it is inferior to the DBLP algorithm, but unlike the latter it can be extended to predicate logic. For various reasons it is interesting to study propositional resolution in its own right.

Consider again propositional formulas in CNF. We write C for a clause, L for a literal and $C \vee L$ instead of $C \cup \{L\}$. Propositional resolution is based on the *resolution rule*

$$\frac{C \vee L \qquad C' \vee \neg L}{C \vee C'}$$

We call the conclusion of a resolution step a *resolvent* of the premises.

Lemma 4.10 (Derivability of Resolution Rule)**.** *Let C, C” be clauses and L a literal. Then $C \vee L, C' \vee \neg L \vdash C \vee C'$.*

Proof. We use the rules of classical propositional natural deduction.

1.	$C \vee L$	hyp
2.	$C' \vee \neg L$	hyp
3.	C	hyp
4.	$C \vee C'$	$\vee I_l$, 3
5.	L	hyp
6.	C'	hyp
7.	$C \vee C''$	$\vee I_r$, 6
8.	$\neg L$	hyp
9.	$\bot$	$\neg E$, 5,8
10.	$C \vee C'$	$\bot E$, 9
11.	$C \vee C'$	$\vee E$, 2,6-7,8-10
12.	$C \vee C'$	$\vee E$, 1,3-4,5-11

□

Alternatively, soundness of the resolution rule can be explained easily by semantic means. See the proof of Lemma 4.13.

Example 4.11. The following examples show simple applications of the resolution rule.

$$\frac{\{P, Q, \neg R\} \qquad \{\neg S, R\}}{\{P, Q, \neg S\}} \qquad\qquad \frac{\{P\} \qquad \{\neg P\}}{\Box}$$

In the first case, $\{P, Q, \neg S\}$ is a resolvent of $\{P, Q, \neg R\}$ and $\{\neg S, R\}$; in the second case, $\Box$ is a resolvent of P and $\neg P$. □

Resolution is suitable for *refutational theorem proving.* To show that a formula φ is a tautology, the formula $\neg\varphi$ is transformed into a CNF formula ψ and resolution is used for deriving the empty clause $\Box$ from ψ. We will see that ψ is unsatisfiable if and only if the empty clause can be derived by resolution, which is the case if and only if the original formula φ is a tautology. But before proving this property we consider an example.

Example 4.12. We show that the clause set $\mathcal{C} = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$ is unsatisfiable. Resolution proofs could be presented like the trees obtained by natural deduction, but in the presence of one single inference rule, the labels associated to the bars of inference rules are not needed. A different tree format is therefore preferable.

Here is a resolution derivation that refutes the clause set.

$$\begin{tikzcd}
\{P, Q\} \arrow[dr, dash] & & \{P, \neg Q\} \arrow[dl, dash] & & \{\neg P, Q\} \arrow[dr, dash] & & \{\neg P, \neg Q\} \arrow[dl, dash] \\
& \{P\} \arrow[drr, dash] & & & & \{\neg P\} \arrow[dll, dash] & \\
& & & \Box & & &
\end{tikzcd}$$

Here is an alternative derivation.

$$\begin{tikzcd}
\{P, Q\} \arrow[dr, dash] \arrow[drrr, dash] & & \{P, \neg Q\} \arrow[dl, dash] & & \{\neg P, Q\} \arrow[dl, dash] & & \{\neg P, \neg Q\} \arrow[ddl, dash] \\
& \{P\} \arrow[ddrr, dash] & & \{Q\} \arrow[drr, dash] & & & \\
& & & & & \{\neg P\} \arrow[dll, dash] & \\
& & & \Box & & &
\end{tikzcd}$$

The resolution algorithm performs proof search in a different way. Starting from the clause set $\mathcal{C}_0 = \mathcal{C}$, it constructs a clause set $\mathcal{C}_1$ to which all possible resolvents from clauses in $\mathcal{C}_0$ have been added. Thus

$$\mathcal{C}_1 = \mathcal{C}_0 \cup \{\{P\}, \{Q\}, \{P, \neg P\}, \{Q, \neg Q\}, \{\neg Q\}, \{\neg P\}\}.$$

In the next step it constructs $\mathcal{C}_2$ to wich all possible resolvents from clauses in $\mathcal{C}_0$ have been added. Thus

$$\mathcal{C}_2 = \mathcal{C}_1 \cup \{\dots, \Box, \dots\}.$$

The empty clause could have been added either as the resolvent of $\{P\}$ and $\{\neg P\}$ or as that of $\{Q\}$ and $\{\neg Q\}$, depending on the order in which $\mathcal{C}_1$ is processed. Proof search can stop at this stage because $\Box$ has been generated.

A derivation tree can be reconstructed if the algorithm remembers the reason why each particular resolvent has been added at some stage. Suppose, for instance, that $\Box$ has been added as a resolvent of $\{P\}$ and $\{\neg P\}$ at stage 2, whereas $\{P\}$ has been added as a resolvent of $\{P, Q\}$ and $\{P, \neg Q\}$, and $\{\neg P\}$ as a resolvent of $\{\neg P, Q\}$ and $\{\neg P, \neg Q\}$ at stage 1. Then we can extract the first derivation tree above from the resolution algorithm. $\Box$

The next lemma shows that adding resolvents to clause sets preserves satisfiability, unsatisfiability and validity.

Lemma 4.13. *If C is a resolvent of two clauses from clause set $\mathcal{C}$, then $\mathcal{C} \equiv \mathcal{C} \cup \{C\}$.*

Proof. If $[\![\mathcal{C} \cup \{C\}]\!] = 1$, then of course $[\![\mathcal{C}]\!] = 1$. For the converse direction suppose $[\![\mathcal{C}]\!] = 1$. Suppose that $C_1 = C_1' \vee L$ and $C_2 = C_2' \vee \neg L$ such that $C = C_!' \cup C_2'$. As clause sets are conjunctions of clauses, it follows that $[\![C_1]\!] = [\![C_2]\!] = 1$.

- If $[\![L]\!] = 0$, then $C_1 \equiv C_1'$, hence $[\![C_1']\!] = 1$ and it follows that $[\![C]\!] = 1$.
- If $[\![L]\!] = 1$, then $C_2 \equiv C_2'$, hence $[\![C_2']\!] = 1$ and it follows that $[\![C]\!] = 1$.

In both cases, therefore $[\![\mathcal{C}]\!] = 1$ implies that $[\![\mathcal{C} \cup \{C\}]\!] = 1$. □

The following definition provides an abstract declarative specification of the resolution algorithm.

Definition 4.14. For every clause set $\mathcal{C}$ let

- $\mathit{res}(X) = X \cup \{C \mid C \textit{ is resolvent of clauses from } X\}$.
- $\mathit{Res}^0(\mathcal{C}) = \mathcal{C}$,
- $\mathit{Res}^{n+1}(\mathcal{C}) = \mathit{res}(\mathit{Res}^n(\mathcal{C}))$,
- $\mathit{Res}^*(\mathcal{C}) = \bigcup_{n \in \mathbb{N}} \mathit{Res}^n(\mathcal{C})$.

The set $\mathit{Res}^*(\mathcal{C})$ is also called *saturation* of the clause set $\mathcal{C}$. It satisfies a number of important properties. First of all, the Res^i form a chain, that is,

$$\mathcal{C} = \mathit{Res}^0(\mathcal{C}) \subseteq \mathit{Res}^1(\mathcal{C}) \subseteq \mathit{Res}^2(\mathcal{C}) \subseteq \cdots \subseteq \mathit{Res}^*(\mathcal{C}).$$

In addition, Res^* is a so-called *closure operator*. It satisfies

$$\mathcal{C} \subseteq \mathit{Res}^*(\mathcal{C}), \qquad \mathit{Res}^*(\mathit{Res}^*(\mathcal{C})) = \mathit{Res}^*(\mathcal{C}), \qquad \mathcal{C} \subseteq \mathcal{C}' \to \mathit{Res}^*(\mathcal{C}) \subseteq \mathit{Res}^*(\mathcal{C}').$$

The second condition, in particular, tells us that one cannot add new resolvents to $\mathit{Res}^*(\mathcal{C})$.

The following theorem shows that the resolution algorithm is sound and refutationally complete.

Theorem 4.15. *A clause set $\mathcal{C}$ is unsatisfiable if and only if $\square \in \mathit{Res}^*(\mathcal{C})$.*

Proof. (*Soundness*). By Lemma 4.13 and induction on n,

$$\mathcal{C} \equiv \mathit{Res}^1(\mathcal{C}) \equiv \cdots \equiv \mathit{Res}^n(\mathcal{C}) \equiv \mathit{Res}^*(\mathcal{C}).$$

Let $\square \in \mathit{Res}^*(\mathcal{C})$. Then $\mathit{Res}^*(\mathcal{C})$ is unsatisfiable and therefore $\mathcal{C}$ is.

(*Refutational Completeness*). Let $\mathcal{C}$ be unsatisfiable. Then, by compactness, some finite $\mathcal{C}_0 \subseteq \mathcal{C}$ must be unsatisfiable. The set $\mathcal{C}_0$ can contain at most n propositional variables $P_1, \ldots, P_n$, for some $n \in \mathbb{N}$. We can thus show that $\Box \in \mathit{Res}^*(\mathcal{C}_0) \subseteq \mathit{Res}^*(\mathcal{C})$ by induction on n and the third closure property.

In the base case, $n = 0$, so that $\mathcal{C}_0 = \{\Box\}$ follows from the assumption of inconsistency (otherwise, if $\mathcal{C}_0 = \emptyset$, then $\mathcal{C}_0$ would be satisfiable).

For the induction step, suppose that $\mathcal{C}_0$ contains $n+1$ variables. Consider the sets

$$\begin{aligned}\mathcal{C}_0^\top &= \mathcal{C}_0[\top/P_{n+1}] = \{C - \{\neg P_{n+1}\} \mid P_{n+1} \notin C \wedge C \in \mathcal{C}_\prime\},\\ \mathcal{C}_0^\bot &= \mathcal{C}_0[\bot/P_{n+1}] = \{C - \{P_{n+1}\} \mid \neg P_{n+1} \notin C \wedge C \in \mathcal{C}_\prime\}.\end{aligned}$$

Both contain n variables because P_{n+1} has been deleted; and both are unsatisfiable. Thus, $\Box \in \mathit{Res}^*(\mathcal{C}_0^\top)$ and $\Box \in \mathit{Res}^*(\mathcal{C}_0^\bot)$ follow from the induction hypothesis.

There is thus a minimal $k \in \mathbb{N}$ for which $\Box \in \mathit{Res}^k(\mathcal{C}_0^\top)$ and we can construct a resolution derivation of $\Box$ with finitely many clauses from $\mathit{Res}^k(\mathcal{C}_0^\top)$ along the lines of Example 4.12. If all of these clauses are in $\mathit{Res}^*(\mathcal{C}_0)$, then we are done and $\Box \in \mathit{Res}^*(\mathcal{C}_0)$. Otherwise, some of them are in $\mathit{Res}^k(\mathcal{C}_0^\top) - \mathit{Res}^*(\mathcal{C}_0)$. Then we construct a derivation of $\{\neg P_{n+1}\} = \Box \cup \{\neg P_{n+1}\}$ in $\mathit{Res}^*(\mathcal{C}_0)$ from the one we have by reinserting $\neg P_{n+1}$ in the clauses from $\mathcal{C}_0^\top$ and all resolvents affected.

We proceed for $\mathit{Res}^*(\mathcal{C}_0^\bot)$ in a similar way with respect to P_{n+1}. Here we either obtain a derivation of $\Box$ or one of $\{P_{n+1}\} = \Box \cup \{P_{n+1}\}$ in $\mathit{Res}^*(\mathcal{C}_0)$.

In the worst case we thus obtain derivations of $\{P_{n+1}\}$ and $\{\neg P_{n+1}\}$ in $\mathit{Res}^*(\mathcal{C}_0)$, that is, $\{P_{n+1}\}, \{\neg P_{n+1}\} \in \mathit{Res}^*(\mathcal{C}_0)$. Hence also $\Box \in \mathit{Res}^*(\mathcal{C}_0)$ by resolution. $\Box$

Example 4.16. In order to illustrate the inductive step in Theorem 4.15, consider again the clause set $\mathcal{C} = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$. In this case,

$$\mathcal{C}^\top = \mathcal{C}[\top/P] = \mathcal{C}^\bot = \mathcal{C}[\bot/P] = \{\{Q\}, \{\neg Q\}\}$$

and obviously, $\Box \in \mathit{Res}^1(\mathcal{C}^\top) = \mathit{Res}^1(\mathcal{C}^\bot)$. We extract the derivation

$$\begin{tikzcd}\{Q\} \arrow[dr, dash] & & \{\neg Q\} \arrow[dl, dash] \\ & \Box & \end{tikzcd}$$

Reinserting $\neg P$ for $\mathcal{C}^\top$ and P for $\mathcal{C}^\bot$ yields the derivations

$$\begin{tikzcd}\{\neg P, Q\} \arrow[dr, dash] & & \{\neg P, \neg Q\} \arrow[dl, dash] & \{P, Q\} \arrow[dr, dash] & & \{P, \neg Q\} \arrow[dl, dash] \\ & \{\neg P\} & & & \{P\} & \end{tikzcd}$$

which can be extended to the first resolution derivation in Example 4.12. $\Box$

Lemma 4.17. *Let $\mathcal{C}$ be a finite clause set. Then $\mathit{Res}^k(\mathcal{C}) = \mathit{Res}^*(\mathcal{C})$ for some $k \in \mathbb{N}$.*

Proof. By finiteness of $\mathcal{C}$, only finitely many propositional variables occur in $\mathcal{C}$. If there are n such variables, then there are $2n$ literals and therefore 2^{2n} clauses The resolution rule does not add new propositional variables. Thus Res^* can add at most 2^{2n} clauses to $\mathcal{C}$ and the resolution algorithm terminates after at most 2^{2n} stages. Thus $Res^*(\mathcal{C}) = Res^{2^{2n}}(\mathcal{C})$ and $k \leq 2^{2n}$. $\square$

It follows that propositional resolution yields yet another solution to the SAT problem.

In practice, resolution is usually combined with redundancy elimination techniques. A typical one is subsumption. A clause C *subsumes* a clause C' if $C \subseteq C'$. Obviously, $C \subseteq C'$ implies $[\![C]\!] \leq [\![C']\!]$ and the following fact is straightforward.

Lemma 4.18. *Let $C, C' \in \mathcal{C}$ with $C \subseteq C'$. Then $\mathcal{C}$ is satisfiable if and only $\mathcal{C} - \{C'\}$ is.*

Proof. Clearly, $\mathcal{C} \models \mathcal{C} - \{C'\}$. For the converse direction let $[\![\mathcal{C} - \{C'\}]\!] = 1$. Then, in particular, $[\![C]\!] = 1 = [\![C']\!]$ and therefore $[\![\mathcal{C}]\!] = 1$. $\square$

Eliminating redundant clauses during the resolution procedure keeps the set $Res^i(\mathcal{C})$ small and can have a crucial effect on performance. It should therefore be applied eagerly.

Example 4.19. Consider once again the set $\mathcal{C} = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$ from Example 4.12. First, the resolvent $\{P\}$ can been added to $\mathcal{C}$. The clauses $\{P, Q\}$ and $\{P, \neg Q\}$ are then subsumed and can be deleted. This yields $\mathcal{C}_1 = \{\{P\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$. Next, the resolvent $\{Q\}$ can be added to $\mathcal{C}_1$. The clause $\{\neg P, Q\}$ is then subsumed. This yields $\mathcal{C}_2 = \{\{P\}, \{Q\}, \{\neg P, \neg Q\}\}$. Next, the resolvent $\{\neg Q\}$ can be added to $\mathcal{C}_2$. The clause $\{\neg P, \neg Q\}$ is then subsumed. This yields the set $\mathcal{C}_2 = \{\{P\}, \{Q\}, \{\neg Q\}\}$. Finally, the resolvent $\square$ can be added to $\mathcal{C}_3$. All clauses in $\mathcal{C}_3$ are then subsumed, and the final set constructed is $\mathcal{C}_4 = \{\square\}$. $\square$

4.3 Clausel Normal Form

Definition 4.20 (Negation Normal Form)**.** A formula of predicate logic is in *negation normal form* if $\leftrightarrow$ and $\rightarrow$ do not occur in it and every negation symbol precedes immediately an atomic formula.

In particular $\neg\neg P(a)$ is not in negation normal form. Every formula can be transformed into an equivalent formula in negation normal form by using the equivalences for CNF normal forms that eliminate $\leftrightarrow$ and $\rightarrow$, and push negations towards atoms. In addition, one can use the quantifier equivalences

$$\neg\forall x.\ \varphi \equiv \exists x.\ \neg\varphi,$$
$$\neg\exists x.\ \varphi \equiv \forall x.\ \neg\varphi$$

for pushing negations towards the leaves of formula trees.

Definition 4.21 (Renamed Formula)**.** A formula of predicate logic is *renamed* if no variable occurs both bound and free in it and different occurrences of quantifiers bind different variables.

Definition 4.22 (Prenex Normal Form)**.** A formula of predicate logic is in *prenex normal form* if it is of the form

$$Q_1 x_1 \dots Q_n x_n.\ \varphi,$$

for some $n \in \mathbb{N}$, where Q_i is either $\forall$ or $\exists$ and φ does not contain any quantifiers.

Theorem 4.23. *Every formula of prediate logic is equivalent to a formula in prenex normal form.*

Proof. By structural induction. □

The proof is algorithmic. We may assume that the formula is already renamed and in negation normal form. The rules

$$\begin{aligned}(\forall x.\ \varphi) \wedge \psi &\equiv \forall x.\ (\varphi \wedge \psi),\\ (\forall x.\ \varphi) \vee \psi &\equiv \forall x.\ (\varphi \vee \psi),\\ (\exists x.\ \varphi) \wedge \psi &\equiv \exists x.\ (\varphi \wedge \psi),\\ (\exists x.\ \varphi) \vee \psi &\equiv \exists x.\ (\varphi \vee \psi),\end{aligned}$$

where x does not occur freely in ψ, helps us to push quantifiers to the top of formula trees. Otherwise we can use the following rules for the same purpose:

$$\begin{aligned}(\forall x.\ \varphi) \wedge (\forall y. \psi) &\equiv \forall x.\ (\varphi \wedge \psi),\\ (\exists x.\ \varphi) \vee (\exists y. \psi) &\equiv \exists x.\ (\varphi \vee \psi)\end{aligned}$$

To summarise, an algorithm for transforming formulas into prenex normal form performs the following steps:

- (Step 1), Rename the formula.
- (Step 2). Transform it into negation normal form.
- (Step 3). Push quantifiers to the top of he formula tree.

Example 4.24. The following examples show how formulas are transformed into prenex normal form.

1.

$$\begin{aligned}(\neg\exists x.\ P(x)) \wedge (\exists y.\ Q(y) \vee \forall z.\ P(z)) &\rightsquigarrow (\forall x.\ \neg P(x)) \wedge \exists y.\ (Q(y) \vee \forall z.\ P(z))\\ &\rightsquigarrow \exists y.\ ((\forall x.\ \neg P(x)) \wedge (Q(y) \vee \forall z.\ P(z)))\\ &\rightsquigarrow \exists y.\ ((\forall x.\ \neg P(x)) \wedge \forall z.\ (Q(y) \vee P(z)))\\ &\rightsquigarrow \exists y \forall z.\ ((\forall x.\ \neg P(x)) \wedge (Q(y) \vee P(z)))\\ &\rightsquigarrow \exists y \forall z \forall x.\ ((\neg P(x)) \wedge (Q(y) \vee P(z)))\end{aligned}$$

2.
$$\begin{aligned}(\forall x.P(x)) \to (\exists y \forall z.\ R(y,z)) &\rightsquigarrow \neg(\forall x.P(x)) \vee (\exists y \forall z.\ R(y,z))\\ &\rightsquigarrow (\exists x.\neg P(x)) \vee (\exists y \forall z.\ R(y,z))\\ &\rightsquigarrow \exists x.\ (\neg P(x) \vee (\forall z.\ R(x,z))\\ &\rightsquigarrow \exists x \forall z.\ (\neg P(x) \vee R(x,z))\end{aligned}$$

After having moved quantifiers to the left, we eliminate them completely. This, however, is not an equivalence transformation.

Definition 4.25 (Skolem Normal Form)**.** A formula of predicate logic is in *Skolem normal form* if it is in prenex normal form and contains only universal quantifiers.

The following lemma states essentially that transformations into Skolem normal forms preserve (un)satisfiability.

Lemma 4.26. *For every sentence φ of predicate logic there exists a sentence $\psi = \forall x_1 \dots \forall y_n.\psi'$ in an extended language with additional function symbols such that $\psi \models \varphi$ and for every model of φ there exists a model expanded by additional function symbols for ψ.*

According to the lemma, new function symbols are needed to compensate for the lack of existential quantifiers. Their introduction is algorithmic. Instead of providing a proof we give an example and a semi-formal argument.

Example 4.27. The formula $\forall m \exists n.\ m < n$ holds in the natural numbers. Obviously there is no fixed number n_0 for which $\forall m.\ m < n_0$ would hold. Instead, n depends on m and is therefore a function of m. We could, for instance, pick $n = m+1$ to eliminate the existential quantifier: $\forall m.\ m < m+1$ holds in $\mathbb{N}$ as well. Yet in more general partial orders we can no longer use this trick. Instead we can replace the formula by $\forall m.m < f(m)$ for a new and therefore arbitrary function f. At least we can always construct a model in wich f is interpreted in such a way that $\forall m.\ m < f(m)$ holds. Therefore, $\forall m \exists n.\ m < n$ holds in the original partial order if and only if $\forall m.\ m < f(m)$ holds in the same partial order, but which a suitable function for f added. □

More generally a Σ-formula $\forall x \exists y.\ \varphi$ is satisfiable if and only if there exists a Σ-model $\mathfrak{A}$ of that formula in which for all $c_1^{\mathfrak{A}} \in A$ there exists a $c_2^{\mathfrak{A}} \in A$ such that $\mathfrak{A} \models \varphi[c_1/x][c_2/y]$. Now expand $\mathfrak{A}$ to a structure $\mathfrak{A}^+$ with carrier set A and a new function $f^{\mathfrak{A}^+} : A \to A$ for which $f^{\mathfrak{A}^+}(c_1^{\mathfrak{A}^+}) = c_2^{\mathfrak{A}^+}$. Expand Σ to Σ^+ by adding the function symbol f that is interpreted by $f^{\mathfrak{A}^+}$ in $\mathfrak{A}^+$. Then, by construction, $\mathfrak{A}^+ \models \forall x.\ \varphi[f(x)/y]$ if and only if course $\mathfrak{A} \models \exists y \forall x.\varphi$.

This argument explains the proof of Lemma 4.26 at least for the $\forall\exists$ quantifier prefix, and the general procedure of replacing existential quantifiers by suitable functions should now be clear. The process of transforming formulas in prenex normal form into Skolem normal form is known as *skolemisation.* In general, we rewrite

$$\forall x_1 \dots \forall x_n \exists y.\ \varphi \rightsquigarrow \forall x_1 \dots \forall x_n.\ \varphi[f(x_1,\dots,x_n)/y],$$

where f is a new function symbol of arity n—a *Skolem function.* For $n = 0$, skolemisation $\exists y.\varphi \rightsquigarrow \varphi[c/y]$ uses, by definition, a 0-ary Skolem function; a *Skolem constant.*

Example 4.28. Skolemising $\exists x.(\neg P(a) \vee P(x))$ yields the formula $\neg P(a) \vee P(c)$ with Skolem constant c. The first formula is valid; the second one only satisfiable. □

This counterexample shows that Skolemisation need not preserve validity. Next we show a simple example of Skolemisation.

Example 4.29. Skolemising $\exists x \forall y \exists z.\ P(x, y, z)$ yields $\forall y.\ P(c, y, f(y))$, where c is a Skolem constant and f a unary Skolem function. □

After skolemisation the universal quantifier prefix can be deleted, that is, left implicit. We can now define the formulas in clause normal form that are the basis of first-order resolution.

Formulas in clause normal form are essentially formulas in conjunctive normal form, but where literals are not propositional variables, but atomic formulas of predicate logic. The following definition is therefore essentially that of CNFs, up to that of literals.

Definition 4.30 (Clause Normal Form)**.** Let Σ be a signature and Φ^A the set of atomic Σ-formulas in Φ except $\bot$ and $\top$.

1. The set of literals over Φ^A is defined, for all $\varphi \in \Phi^A_\Sigma$, by $\mathcal{L} ::= \varphi \mid \neg\varphi$.
2. The set of *clauses* over Φ^A is defined by $\mathcal{D} ::= \bot \mid \mathcal{L} \mid \mathcal{L} \vee \mathcal{D}$.
3. The set of formulas in *clause normal form* over Φ^A is defined $\mathcal{C} ::= \top \mid \mathcal{D} \mid \mathcal{C} \wedge \mathcal{D}$.

To summarise, the algorithm for transforming sentences of predicate logic into clause normal form has the following steps:

- (Step 1) Bring formulas in prenex form.
- (Step 2) Skolemise and eliminate universal quantifiers.
- (Step 3) Transform in CNF.

By Lemma 4.26 and previous results on prenex normal forms and CNF, the output of the algorithm is a formula that is (un)satisfiable if and only if the input formula is, but not necessarily equivalent. This is sufficient for refutational theorem proving.

4.4 Proof Search for Predicate Logic

This section explains how the resolution algorithm can be extended to predicate logic. Obviously, if clause sets contain only *ground clauses*, that is, clauses without any free variables, then we can use propositional resolution for refutational theorem proving anyway.

Otherwise, for formulas in clause normal form, the basic idea is to guess the appropriate ground instances to perform propositional resolution. First we show that algorithm works in principle, but is inefficient in practice. Then we show how it can be improved by instantiating clauses only as far as needed.

4.4.1 Ground Resolution

Definition 4.31. A non-empty Σ structure $\mathfrak{H}$ is called *Herbrand* structure if

1. the carrier set H is the set of all ground Σ-terms,
2. $f^{\mathfrak{H}}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$ holds for all $f \in \Sigma$ and $t_!, \dots, t_n \in H$.

In a Herbrand structure, therefore, terms are interpreted simply as themselves. The interpretation of terms is therefore fixed; only the interpretation of predicate symbols can vary.

Example 4.32. If $\Sigma = \{f, c\}$ for a unary function symbol f and a constant c, then $H = \{a, \} \cup \{f^i(a) \mid i \geq 1\}$, $c^{\mathfrak{H}} = c$ and $f^{\mathfrak{H}}(f^n(c)) = f^{n+1}(c)$. $\square$

A simple structural induction shows that $[\![t]\!]^{\mathfrak{H}} = t$ holds indeed for any term t in a Herbrand structure $\mathfrak{H}$.

This may lead to an infinite set, but our soundness and refutational completeness result for resolution still applies! If the initial clause set $\mathcal{C}$ is unsatisfiable, one is still guaranteed to find the empty clause $\square$ in $Res^*(\mathcal{C}'$, where $\mathcal{C}'$ is the set of all ground instances of $\mathcal{C}$, within finitely many steps. Otherwise, however, if $\mathcal{C}$ is satisfiable, the algorithm may search forever. Thus resolution for propositional logic is only a semidecision procedure.

Theorem 4.33 (Herbrand's Theorem)**.** *Let $\forall x_1, \dots, x_n.\ \varphi$ be a sentence in Skolem normal form (with φ variable-free). Then φ is satisfiable if and only if it has a Herbrand model.*

Proof. If φ has a Herbrand model, then it is of course satisfiable. For the converse direction, suppose that φ has a model $\mathfrak{A}$. We need to fix the interpretation of predicate symbols to construct a Herbrand model $\mathfrak{H}$ of φ. This is achieved simply by stipulating

$$(t_1, \dots, t_n) \in P^{\mathfrak{H}} \Leftrightarrow \mathfrak{A} \models P(t_1, \dots, t_n).$$

We show by induction on n that $\mathfrak{A} \models \forall x_1, \dots, x_n.\ \varphi$ implies $\mathfrak{H} \models \forall x_1, \dots, x_n.\ \varphi$.

- If $n = 0$, then φ is a ground formula in clause normal form. By the above interpretation of predicate symbols, $\mathfrak{A} \models \varphi \Leftrightarrow \mathfrak{H} \models \varphi$.
- For the induction step, we have the sentence $\forall x.\ \varphi$. We know that $\mathfrak{A} \models \forall x.\ \varphi$ if and only if $\mathfrak{A} \models \varphi[c/x]$ for all $c^{\mathfrak{A}} \in A$. Since $\varphi[c/x]$ is a sentence, the induction hypothesis yields $\mathfrak{H} \models \varphi[t/x]$ for all $t \in H$, whence $\mathfrak{H} \models \forall x . \varphi$.

The claim is then obvious. $\square$

Definition 4.34 (Herbrand Expansion)**.** The *Herbrand expansion* of a formula $\forall x_1, \dots x_n.\ \varphi$ in Skolem normal form is the set

$$\eta(\forall x_1, \dots x_n.\ \varphi) = \{\varphi[t_1/x_1] \dots [t_n/x_n] \mid t_1, \dots, t_n \text{ are ground terms}\}.$$

Lemma 4.35. *A formula φ in Skolem normal form is satisfiable if and only if $\eta(\varphi)$ is satisfiable.*

Proof. φ is satisfiable if and only if it has a Herbrand model. The claim then follows from the definition of satisfiability of universal formulas. $\square$

Theorem 4.36 (Ground Resolution)**.** *A formula φ in clause normal form is satisfiable if and only if $\square \in Res^*(\eta(\varphi))$.*

Proof. By the previous lemma and soundness and completeness of propositional resolution. $\square$

Example 4.37. The following refutation is obtained by guessing suitable ground instances in the second row of the derivation.

$$\begin{tikzcd}
\{P(z_1,z_1),\neg Q(z_2)\} \arrow[d, dash] & & & \{\neg P(a,y)\} \arrow[dl, dash] \arrow[dr, dash] & & & \{P(x_1,b),Q(f(x_1,x_2))\} \arrow[d, dash] \\
\{P(a,a),\neg Q(f(a,b))\} \arrow[drr, dash] & & \{\neg P(a,a)\} \arrow[d, dash] & & \{\neg P(a,b)\} \arrow[d, dash] & & \{P(a,b) \vee Q(f(a,b))\} \arrow[dll, dash] \\
& & \{\neg Q(f(a,b))\} \arrow[dr, dash] & & \{Q(f(a,b))\} \arrow[dl, dash] & & \\
& & & \square & & &
\end{tikzcd}$$

The situation can be summarised as follows. The ground resolution theorem shows that propositional resolution is essentially refutationally complete for predicate logic, but on a satisfiable formula proof search may proceed forever.

For a machine it is of course impossible to guess suitable instances. Nevertheless, for any given formula in clause normal form, one can build Herbrand carrier set H from the constant and function symbols that appear in the formula (if there are none, then just add constant a). The Herbrand universe is clearly countably infinite and one can enumerate the Herbrand expansion of a given formula step by step and build resolution closures for each iteration.

4.4.2 Unification

The question remains how far one needs to instantiate clauses to perform resolution proofs. If that is possible, one can potentially represent large parts of Herbrand universes by partial instantiation.

Example 4.38. The following refutation is obtained by instantiating formulas as far as needed for resolution proofs.

$$\begin{tikzcd}
\{P(z_1, z_1), \neg Q(z_2)\} \arrow[d, dash] & & \{\neg P(a, y)\} \arrow[ddl, dash] \arrow[dr, dash] & & \{P(x_1, b), Q(f(x_1, x_2))\} \arrow[d, dash] \\
\{P(a, y), \neg Q(z_2)\} \arrow[dr, dash] & & & \{\neg P(a, b)\} \arrow[d, dash] & \{P(a, b) \vee Q(f(a, x_2))\} \arrow[dl, dash] \\
& \{\neg Q(z_2)\} \arrow[d, dash] & & \{Q(f(a, x_2))\} \arrow[ddl, dash] & \\
& \{\neg Q(f(a, x_2))\} \arrow[dr, dash] & & & \\
& & \square & &
\end{tikzcd}$$

□

The idea is thus to apply resolution rules

$$\frac{C \vee L \qquad C' \vee \neg L'}{(C \vee C')\sigma}$$

where σ is a substitution for which $L\sigma = L'\sigma$ and $C \vee L$ and $C' \vee \neg L'$ are renamed (it is customary in the field of resolution to write $C\sigma$ instead of $\sigma(C)$ and we follow this tradition in this section). The question remains whether and how we can compute most general instances.

Definition 4.39 (Unifier)**.** Let s and t be terms.

1. A substitution σ is a *unifier* of s and t if $s\sigma = t\sigma$.
2. A unifier σ is a *most general unifier* (*mgu*) of s and t if for all unifiers ρ of s and t there exists a substitution τ such that $\rho = \tau \circ \sigma$.

Lemma 4.40. *Two unifiable terms have a most general unifier.*

Example 4.41. The substitution $\sigma : x \mapsto g(a, b), y \mapsto g(g(a, b), b), z \mapsto c$ is a mgu of the terms $f(g(x, b), f(x, z))$ and $f(y, f(g(a, b), c)$.

$$\begin{aligned} f(g(x, b), f(x, z))[g(a, b)/x][c/z] &= f(g(g(a, b), b), f(g(a, b), c)) \\ &= f(y, f(g(a, b), c))[g(g(a, b), b)/y]. \end{aligned}$$

□

An algorithmic approach to unification requires a slight generalisation, namely to find a simultaneous most general unifier for a set $s_1 \approx t_!, \dots s_n \approx t_n$ of pairs of terms. We write E for such a set and abbreviate $E \cup \{s \approx t\}$ as $E, s \approx t$.

Lemma 4.42. *The following algorithm computes the most general unifier of a set of terms:*

$$
\begin{aligned}
E, s \approx s &\rightsquigarrow E, && \\
E, f(s_1, \dots, s_n) \approx f(t_1, \dots t_n) &\rightsquigarrow E, s_1 \approx t_!, \dots s_n \approx t_n, && \\
E, f(\dots) \approx g(\dots) &\rightsquigarrow \bot, && \\
E, t \approx x &\rightsquigarrow E, x \approx t, && \text{if } t \notin x, \\
E, x \approx t &\rightsquigarrow \bot, && \text{if } x \neg t \text{ and } x \in V(t), \\
E, x \approx t &\rightsquigarrow E[t/x], x \approx t, && \text{if } x \notin V(t).
\end{aligned}
$$

If the initial set is not unifier, then the algorithm terminates with $\bot$. Otherwise, it computes a set $x_1 \approx t_1, \dots x_n \approx t_n$ representing the most general unifier $x_1 \mapsto t_1, \dots x_n \mapsto t_n$

The proof of correctness and termination of this algorithm is beyond the scope of these lectures.

To compute the most general unifier of two terms s and t, the above unification algorithm is run on the initial set $s \approx t$. We write $mgu(s, t)$ for the most general unifier of s and t.

Example 4.43.

1.
$$
\begin{aligned}
f(g(x, b), f(x, z)) \approx f(y, f(g(a, b), c) &\rightsquigarrow g(x, b) \approx y, f(x, z) \approx f(g(a, b), c) \\
&\rightsquigarrow y \approx g(x, b), x \approx g(a, b), z \approx c \\
&\rightsquigarrow y \approx g(g(a, b), b), x \approx g(a, b), z \approx c
\end{aligned}
$$

2. $f(x, b) \approx f(a, y) \rightsquigarrow x \approx a, b \approx y \rightsquigarrow x \approx a, y \approx b$

3. $f(x, x) \approx f(a, b) \rightsquigarrow x \approx a, x \approx b \rightsquigarrow a \approx b, x \approx b \rightarrow \bot$

4.
$$
\begin{aligned}
f(x, g(y)) \approx f(y, x) &\rightsquigarrow x \approx y, g(y) \approx x \\
&\rightsquigarrow x \approx y, x \approx g(y) \\
&\rightsquigarrow g(y) \approx y, x \approx g(y) \\
&\rightsquigarrow y \approx g(y), x \approx g(y) \\
&\rightsquigarrow \bot
\end{aligned}
$$

□

4.4.3 Resolution for Predicate Logic

It has been one of the most significant insights of proof search for predicate logic that the combination of resolution and unification makes the method scale.

The rule of resolution for predicate logic is

$$\frac{C \lor L \qquad C' \lor \neg L'}{(C \lor C')\sigma} \qquad \sigma = mgu(L, L').$$

Note that the premises need to be renamed.

Example 4.44. $\mathcal{C} = \{\{P(x), P(y)\}, \{\neg P(a), \neg P(b)\}\}$ is clearly unsatisfiable. But one can readily convince oneself that $Res^*(\mathcal{C})$ does not contain clauses with less than two literals. $\Box$

A second inference rule, called *factoring rule*, is therefore needed:

$$\frac{C \lor L \lor L'}{(C \lor C')\sigma} \qquad \sigma = mgu(L, L').$$

Alternatively, one can use the generalise resolution rule

$$\frac{C \lor L_1 \lor \cdots \lor L_m \qquad C' \lor \neg L_1' \lor \cdots \lor L_n'}{(C \lor C')\sigma} \qquad \sigma = mgu(L_1, \ldots, L_m, L_1', \ldots, L_n').$$

This is simpler to use in soundness and completeness proofs.

Example 4.45. Negating the formula $H(s), \forall x.\ (H(x) \to M(x)) \vdash M(s)$ and transforming it into clause normal form yields

$$\{\{H(s)\}, \{\neg H(x), M(x)\}, \{\neg M(s)\}\}.$$

We can then use resolution to derive $\Box$:

$$\begin{tikzcd}
\{H(s)\} \arrow[dr, dash] & & \{\neg H(x), M(x)\} \arrow[dl, dash] & & \{\neg M(s)\} \arrow[ddll, dash] \\
& \{M(s)\} \arrow[dr, dash] & & & \\
& & \Box & &
\end{tikzcd}$$

The mgu used is $x \mapsto s$. The original formula is thus valid. $\Box$

Example 4.46. Negating the formula $\exists y \forall x.\ P(x, y) \to \forall x \exists y.\ P(x, y)$ and transforming it into clause normal yields $\{\{P(x, a), \neg P(f(x), w)\}$. We can then use resolution to derive $\Box$:

$$\begin{tikzcd}
\{P(x, a)\} \arrow[dr, dash] & & \{\neg P(f(y), w)\} \arrow[dl, dash] \\
& \Box &
\end{tikzcd}$$

The mgu used is $x \mapsto f(y), w \mapsto a$. The original formula is thus valid. $\Box$

Example 4.47 (Drinker Paradox)**.** The formula $\exists x.\ (P(x) \to \forall y.P(y))$ is quite hard to prove with natural deduction. Negating it and transforming into clause normal form yields $\{\{P(x)\}, \{\neg P(f(y))\}\}$. Then

$$\begin{tikzcd}[column sep=small]
\{P(x)\} \arrow[dr, no head] & & \{\neg P(f(y))\} \arrow[dl, no head] \\
& \square &
\end{tikzcd}$$

with mgu $x \mapsto f(y)$. The original formula is thus valid. $\square$

Finally we turn to soundness and completness of resolution for predicate logic.

Theorem 4.48 (Soundness)**.** *Let $\mathcal{C}$ be a formula in clause normal form, for which $\square \in$ $Res^*(\mathcal{C})$. Then $\mathcal{C}$ is unsatisfiable.*

The proof is similar to that of propositional logic. For completeness, the question remains whether all ground instances of resolution steps are instances of non-ground ones.

Lemma 4.49 (Lifting Lemma)**.** *Let $C_!$ and C_2 be clauses with disjoint variables and ground instances C_1' and C_2' tha can be resolved to C' by ground resolution. Then C_1 and C_2 can be resolved to C and C' is an instance of C.*

Theorem 4.50 (Refutational Completeness)**.** *Let $\mathcal{C}$ be an unsatisfiable formula in clause normal form. Then $\square \in Res^*(\mathcal{C})$ with respect to generalised resolution.*

Proof. Let $\mathcal{C}$ be unsatisfiable. Then $\square \in Res^*(\mathcal{C}")$ for a set of ground instances $\mathcal{C}'$ of $\mathcal{C}$ and one can extract a resolution derivation from this. Then the lifting lemma allows us to extract a non-ground proof. $\square$

As we have seen in Chapter 3, equality in an important predicate. The efficient treatment of equality in refutational theorem proving has been a prominent line of research over the last three decades. Roughly speaking, resolution-based theorem proving with equality combines logic with powerful rewriting procedures that have been developed, for instance, for computer algebra systems.

Isabelle/HOL, for instance, calls several automated theorem provers through its Sledgehammer tactic, and reconstructs their proofs internally to increase trustworthiness. This often allows to prove mathematical theorems automatically that humans would find hard.

Example 4.51 (Robbins' Conjecture)**.** A *Robbins algebra* is a structure $(R, +, ')$ that satisfies the axioms

$$x + (y + z) = (x + y) + z, \qquad x + y = y + x, \qquad ((x + y)' + (x' + y)')' = x.$$

If one replaces the third axiom by $(x'+y)'+(x'+y')' = x$, which is called Huntington's axiom, then one obtains (Huntington's axiomatisation of) boolean algebras, that is, the standard axioms of boolean algebra are derivable from the first two axioms above plus Huntington's axiom.

The question is whether every Robbins algebra is a boolean algebra, too. It has been open since around 1935 and was solved by an automated theorem prover in 1996. At that time, the proof search took about eight days. □

Chapter 5

Temporal Logics

A main application of logic in computer science is the verification of programs and computing systems. This chapter gives a glimpse at a state-of-the-art verification technology called *model checking*, which is by now widely used in industrial practice.

We have already seen that databases can be modelled as finite relational structures, and database queries are related to logical formulas that hold in that structure: $\mathfrak{D} \models \varphi$. The same approach can be applied to other computing systems, so long as they can be represented by finite relational structures. If a structure $\mathfrak{A}$ models a system, and a formula φ describes the desired behaviour of such a system, then

$$\mathfrak{A} \models \varphi$$

holds if the behaviour expressed by φ holds in the system modelled by $\mathfrak{A}$.

Typical applications of model checking include the verification of protocols, hardware systems and, in particular, reactive systems such as control and operating systems, where termination implies failure.

Model checking technology includes specification languages for modelling the system $\mathfrak{A}$. These are often similar to programming languages. For expressing the systen properties φ to be checked, temporal logics are widely used. These allow one to specify that bad properties never hold or that desirable properties hold infinitely often. Finally, efficient algorithms are needed to check whether properties expressed by φ hold in a model $\mathfrak{A}$.

In practice, a model checker usually returns "True" if φ holds in $\mathfrak{A}$, and it returns a counterexample for debugging otherwise.

A typical example is a *mutual exclusion* protocol from concurrency control. Two or more agents or processors share some resource such as a printer, a global variable, or a database in the cloud. This shared resource is known as the *critical section*. The protocol is assumed to satisfy four main correctness properties:

1. *Safety*: At most one process must be in the critical section at any time.

2. *Liveness*: If a process requests to enter the critical section, then this request will eventually be granted.

3. *Non-blocking*: Any process can always request to enter the critical section.

4. *No strict sequencing*: Processes need not enter the critical section alternatingly.

Other properties are that no process can stay in the critical section forever (*no deadlock*) and that if a process requests infinitely often to enter the critical section, then it will be granted access infinitely often (*fairness*).

In the next sections we will see how so-called labelled transition properties can be used for modelling computing systems and how a temporal logic can be used for modelling their behaviour.

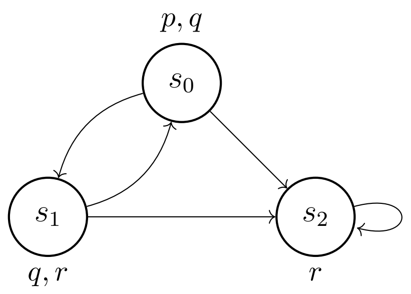
5.1 Transition Systems

Definition 5.1 (Labelled Transition System)**.** Let P be a set of propositional variables. A *labelled transition system* (*LTS*) is a structure $(S, \to, \lambda)$ where S is a finite set of states, $\to \subseteq S \times S$ a binary relation—a *transition relation*—and $\lambda : S \to \mathcal{P}(P)$ a labelling function, which labels states with propositional variables.

We generally require that for all $s \in S$ there exists a $s' \in S$ such that $s \to s'$.

Labelled transition systems are best visualised as directed graphs.

Example 5.2. The graph



represents the LTS $(S, \to, \lambda)$ with

- $S = \{s_0, s_1, s_2\}$,
- $\to = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$,
- $\lambda : s_0 \mapsto \{p, q\}, s_1 \mapsto \{q, r\}, s_2 \mapsto \{r\}$.

□

A *run* or *trace* of a LTS is represented by a path through it.

Definition 5.3. A *path* in a LTS $(S, \to, \lambda)$ is a sequence $\pi : \mathbb{N} \to S$ such that $s_i \to s_{i+1}$ holds for all $i \in \mathbb{N}$.

We write $\pi = s_0 s_1 s_2 \ldots$ and $\pi^i = s_i s_{i+1} s_{i+2} \ldots$ for the ith *suffix* of π.

The labelling function of a LTS tells us which propositions hold in the states of an LTS. In the above example, for instance, p and q hold in s_0, whereas r does not hold. q and r hold in s_1, whereas p does not hold. The labelling function thus makes properties change along a path, dynamically with time.

$$\begin{tikzcd}
\overset{p,q}{s_0} \arrow[r] & \overset{q,r}{s_1} \arrow[r] & \overset{p,q}{s_0} \arrow[r] & \overset{q,r}{s_1} \arrow[r] & \overset{r}{s_2} \arrow[r] & \overset{r}{s_2} \arrow[r, dashed] & {}
\end{tikzcd}$$

Linear temporal logic can be used for analysing properties along such infinite paths.

5.2 Linear Temporal Logic

Definition 5.4. Let P be a set of propositional variables. The formulas of *linear temporal logic* (*LTL*) are defined for $p \in P$, by the syntax

$$\Phi ::= \bot \mid \top \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \to \Phi \mid X\Phi \mid F\Phi \mid G\Phi \mid \Phi U \Phi \mid \Phi W \Phi.$$

- $\bot$, $\top$, $\wedge$, $\vee$ and $\to$ are the usual propositional connectives;
- X, F, G, U and W are *temporal connectives* or *temporal modalities.*

The intuitive meaning of the temporal connectives on a path π is as follows.

- $X\varphi$ means that φ holds in the *next* state on a path π; that is, on π^1,
- $F\varphi$ means that φ holds *eventually* along π; that is, on some suffix π^k,
- $G\varphi$ means that φ holds *always* along π, that is, on all suffices π^k,
- $\varphi U \psi$ means that φ holds along π *until* ψ holds on some suffix π^k,
- $\varphi W \psi$ means that either $\varphi U \psi$ holds along π, or else $G\varphi$.

For this reason, $X\varphi$ is pronounced "*next* φ", $F\varphi$ "*eventually* φ", $G\varphi$ "*globally* φ" and $\varphi U \psi$ "φ *until* ψ". Finally. W is the *weak until* modality.

Next we define the semantics of LTL.

Definition 5.5. Let $(S, \to, \lambda)$ be a LTS and $\pi = s_0 s_1 \ldots$ a path. The *satisfiability relation* $\pi \models \varphi$ is defined recursively over the LTL syntax by

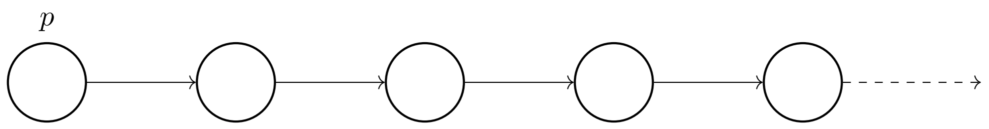
- $\pi \not\models \bot$ and $\pi \models \top$,
- $\pi \models p$ if and only if $p \in \lambda(s_0)$,
- $\pi \models \neg\varphi$ if and only of $\pi \not\models \varphi$,

- $\pi \models \varphi \wedge \psi$ if and only if $\pi \models \varphi$ and $\pi \models \psi$,
- $\pi \models \varphi \vee \psi$ if and only if $\pi \models \varphi$ or $\pi \models \psi$,
- $\pi \models \varphi \to \psi$ if and only if $\pi \models \psi$ whenever $\pi \models \varphi$,
- $\pi \models X\varphi$ if and only if $\pi^1 \models \varphi$,
- $\pi \models F\varphi$ if and only if $\pi^k \models \varphi$ for some $k \in \mathbb{N}$,
- $\pi \models G\varphi$ if and only if $\pi^k \models \varphi$ for all $k \in \mathbb{N}$,
- $\pi \models \varphi U \psi$ if and only if $\pi^k \models \psi$ for some $k \in \mathbb{N}$ and $\pi \models \psi$ for all $0 \leq i < k$,
- $\pi \models \varphi W \psi$ if and only if either $\pi j \models \psi$ for all $j \in \mathbb{N}$, or $\pi^k \models \psi$ for some $k \in \mathbb{N}$ and $\pi \models \psi$ for all $0 \leq i < k$.

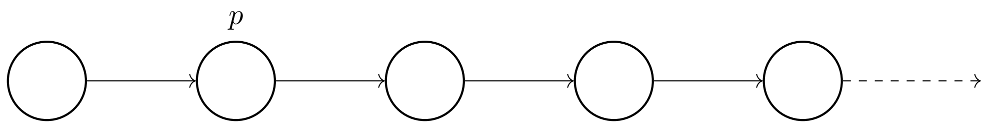
This definition is probably best explained by example

Example 5.6.

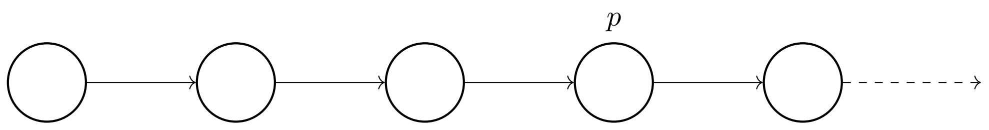
1. $\pi \models p$.



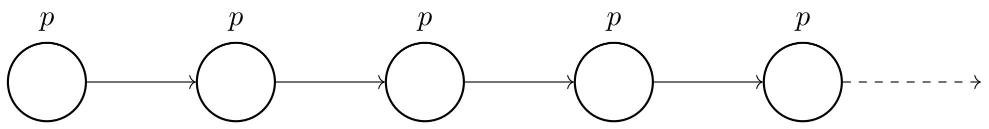
2. $\pi \models Xp$.



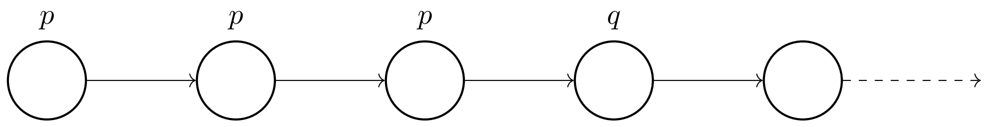
3. $\pi \models Fp$.



4. $\pi \models Gp$.



5. $\pi \models pUq$.



$\square$

Like in classical propositional logic, it turns out that not all temporal operators are needed, that is, some of them are redundant. In fact, the following reduced syntax suffices:

$$\Phi ::= \bot \mid p \mid \neg\Phi \mid \Phi \vee \Phi \mid X\Phi \mid \Phi U \Phi.$$

Definition 5.7. The LTL formulas φ and ψ are *equivalent*, $\varphi \equiv \psi$, if $\pi \models \varphi$ if and only if $\pi \models \psi$ holds for all paths π in all LTS's $\mathfrak{S}$.

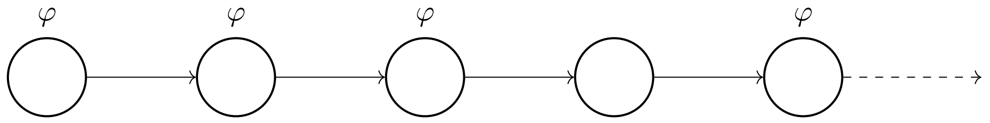
Lemma 5.8. *The following LTL formulas are equivalent:*

1. $\neg X\varphi \equiv X\neg\varphi$,
2. $\neg G\varphi \equiv F\neg\varphi$,
3. $\neg F\varphi \equiv G\neg\varphi$,
4. $F(\varphi \vee \psi) \equiv F\varphi \vee F\psi$,
5. $G(\varphi \wedge \psi) \equiv G\varphi \wedge G\psi$,
6. $F\varphi \equiv \top U\varphi$,
7. $G\varphi \equiv \neg(\top U\neg\varphi)$,
8. $\varphi U\psi \equiv \psi \vee (\varphi \wedge X(\varphi U\psi))$.

Proof. We prove (2) as an example. Fix a path π in the LTS $\mathfrak{S}$. Then

$$\begin{aligned}\pi \models \neg G\varphi &\Leftrightarrow G \not\models \varphi\\ &\Leftrightarrow \text{it is not the case that } \pi^i \not\models \varphi \text{ holds for all } i \in \mathbb{N}\\ &\Leftrightarrow \text{there is some } i \in \mathbb{N} \text{ such that } \pi^i \not\models \varphi\\ &\Leftrightarrow \text{there is some } i \in \mathbb{N} \text{ such that } \pi^i \models \neg\varphi\\ &\Leftrightarrow \pi \models F\neg\varphi.\end{aligned}$$

Alternatively one can describe the situation by using the following graph for π.

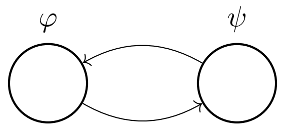


It is not the case that $G\varphi$ holds along π if and only if some state of π is not labelled with p. This is the case if and only if $F\neg\varphi$ holds along the path.

The other properties can be verified by similar means. □

The equivalence (8) describes $\varphi U\psi$ from an operational point of view. To check that this formula holds in the current state one must either have ψ in this state, or else φ must hold and one must continue to verify $\varphi U\psi$ in the next state.

Example 5.9. To check that $G(\varphi \vee \psi) \not\equiv G\varphi \vee G\psi$, one must find a counterexample. Consider the LTS



Then φ and ψ hold alternatingly along each path and therefore $\pi \models G(\varphi \vee \psi)$ holds for all paths π. However neither φ nor ψ holds everywhere along any path, and therefore $\pi \not\models \varphi$ and $\pi \not\models \psi$ for any path π. Hence no path π satisfies $\pi \models G\varphi \vee G\psi$. □

5.3 Model Checking: A Taste of Verification

We can now give a mathematical definition of the model checking problem.

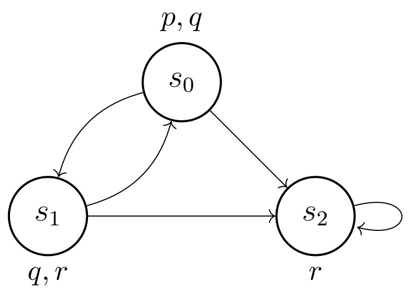
Definition 5.10 (Model Checking Problem)**.** Let $\mathfrak{S} = (S, \to, \lambda)$ be a LTS and φ a LTL formula. The *model checking problem*

$$\mathfrak{S}, s \models \varphi$$

is the problem of deciding $\pi \models \varphi$ for all paths π of $\mathfrak{S}$ with $s = s_0$.

We often write $s \models \varphi$ when the LTS is fixed.

Example 5.11. Consider again the LTS



from Example 5.2. The following formulas hold.

- $s_0 \models p \wedge q$ and $s_0 \models \neg r$,

- $s_0 \models Xr$,
- $s_0 \not\models Xq$,
- $s_0 \models G\neg(p \wedge r)$,
- $s_2 \models Gr$,
- $s_i \models F\neg q \rightarrow FGr$, for $i \in \{0, 1, 2\}$,
- $s_0 \models GFp \rightarrow GFr$,
- $s_0 \not\models GFr \rightarrow Gfp$.

□

While LTL allows arbitrary nestings of temporal operators, a few specification pattern cover many system properties that appear in practice. Here we list some of these practical specification pattern.

- It is never the case that the processes p_1 and p_2 are at the same time at the critical section cs_1 and cs_2 (safety).
$$G\neg(cs_1 \wedge cs_2).$$
- Every request will eventually be granted (liveness).
$$G(\mathit{request} \rightarrow F\,\mathit{granted}).$$
- A process will be enabled infinitely often (liveness).
$$GF\,\mathit{enabled}.$$
- A process that is enabled infinitely often will run infinitely often (fairness).
$$GF\,\mathit{enabled} \rightarrow GF\,\mathit{run}.$$
- A process will not become permanently inactive in the future (deadlock freedom).
$$\neg FG\neg \mathit{active}.$$

Some properties can, however, not be expressed with LTL. These a typically properties that hold along a particular part of a system. It cannot, for instance, be expressed that a process can always request to enter the critical section. Such properties require other temporal logics that are beyond the scope of these lectures.

Real-world LTL model checking is based on efficient search algorithms and constructions on automata that accept infinite strings. These are once again beyond the scope of these lectures.