LOGIC IN COMPUTER SCIENCE

Modelling and Reasoning about Systems

MICHAEL HUTH
Department of Computing
Imperial College London, United Kingdom

MARK RYAN
School of Computer Science
University of Birmingham, United Kingdom

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521543101

First published in print format 2004

ISBN-13 978-0-511-26401-6 eBook (EBL)
ISBN-10 0-511-26401-1 eBook (EBL)

ISBN-13 978-0-521-54310-1 paperback
ISBN-10 0-521-54310-X paperback

Contents

Foreword to the first edition

by

Edmund M. Clarke
FORE Systems Professor of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Formal methods have finally come of age! Specification languages, theorem provers, and model checkers are beginning to be used routinely in industry. Mathematical logic is basic to all of these techniques. Until now textbooks on logic for computer scientists have not kept pace with the development of tools for hardware and software specification and verification. For example, in spite of the success of model checking in verifying sequential circuit designs and communication protocols, until now I did not know of a single text, suitable for undergraduate and beginning graduate students, that attempts to explain how this technique works. As a result, this material is rarely taught to computer scientists and electrical engineers who will need to use it as part of their jobs in the near future. Instead, engineers avoid using formal methods in situations where the methods would be of genuine benefit or complain that the concepts and notation used by the tools are complicated and unnatural. This is unfortunate since the underlying mathematics is generally quite simple, certainly no more difficult than the concepts from mathematical analysis that every calculus student is expected to learn.

Logic in Computer Science by Huth and Ryan is an exceptional book. I was amazed when I looked through it for the first time. In addition to propositional and predicate logic, it has a particularly thorough treatment of temporal logic and model checking. In fact, the book is quite remarkable in how much of this material it is able to cover: linear and branching time temporal logic, explicit state model checking, fairness, the basic fixpoint

theorems for computation tree logic (CTL), even binary decision diagrams and symbolic model checking. Moreover, this material is presented at a level that is accessible to undergraduate and beginning graduate students. Numerous problems and examples are provided to help students master the material in the book. Since both Huth and Ryan are active researchers in logics of programs and program verification, they write with considerable authority.

In summary, the material in this book is up-to-date, practical, and elegantly presented. The book is a wonderful example of what a modern text on logic for computer science should be like. I recommend it to the reader with greatest enthusiasm and predict that the book will be an enormous success.

(This foreword is re-printed in the second edition with its author's permission.)

Preface to the second edition

Our motivation for (re)writing this book

One of the leitmotifs of writing the first edition of our book was the observation that most logics used in the design, specification and verification of computer systems fundamentally deal with a *satisfaction relation*

$$\mathcal{M} \models \phi$$

where $\mathcal{M}$ is some sort of *situation* or *model* of a system, and ϕ is a specification, a formula of that logic, expressing what should be true in situation $\mathcal{M}$. At the heart of this set-up is that one can often specify and implement algorithms for computing $\models$. We developed this theme for propositional, first-order, temporal, modal, and program logics. Based on the encouraging feedback received from five continents we are pleased to hereby present the second edition of this text which means to preserve and improve on the original intent of the first edition.

What's new and what's gone

Chapter 1 now discusses the design, correctness, and complexity of a SAT solver (a marking algorithm similar to Stålmarck's method [SS90]) for full propositional logic.

Chapter 2 now contains basic results from model theory (Compactness Theorem and Löwenheim–Skolem Theorem); a section on the transitive closure and the expressiveness of existential and universal second-order logic; and a section on the use of the object modelling language Alloy and its analyser for specifying and exploring under-specified first-order logic models with respect to properties written in first-order logic with transitive closure. The Alloy language is executable which makes such exploration interactive and formal.

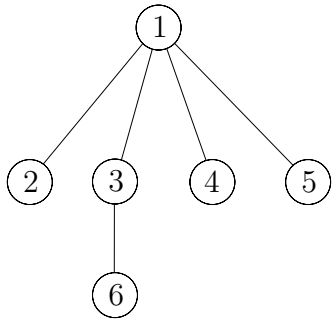
Chapter 3 has been completely restructured. It now begins with a discussion of linear-time temporal logic; features the open-source NuSMV model-checking tool throughout; and includes a discussion on planning problems, more material on the expressiveness of temporal logics, and new modelling examples.

Chapter 4 contains more material on total correctness proofs and a new section on the programming-by-contract paradigm of verifying program correctness.

Chapters 5 and 6 have also been revised, with many small alterations and corrections.

The interdependence of chapters and prerequisites

The book requires that students know the basics of elementary arithmetic and naive set theoretic concepts and notation. The core material of Chapter 1 (everything except Sections 1.4.3 to 1.6.2) is essential for all of the chapters that follow. Other than that, only Chapter 6 depends on Chapter 3 and a basic understanding of the static scoping rules covered in Chapter 2 – although one may easily cover Sections 6.1 and 6.2 without having done Chapter 3 at all. Roughly, the interdependence diagram of chapters is



WWW page

This book is supported by a Web page, which contains a list of errata; text files for all the program code; ancillary technical material and links; all the figures; an interactive tutor based on multiple-choice questions; and details of how instructors can obtain the solutions to exercises in this book which are marked with a $*$. The URL for the book's page is `www.cs.bham.ac.uk/research/lics/`. See also `www.cambridge.org/052154310x`

Acknowledgements

Many people have, directly or indirectly, assisted us in writing this book. David Schmidt kindly provided serveral exercises for Chapter 4. Krysia Broda has pointed out some typographical errors and she and the other authors of [BEKV94] have allowed us to use some exercises from that book. We have also borrowed exercises or examples from [Hod77] and [FHMV95]. Susan Eisenbach provided a first description of the Package Dependency System that we model in Alloy in Chapter 2. Daniel Jackson make very helpful comments on versions of that section. Zena Matilde Ariola, Josh Hodas, Jan Komorowski, Sergey Kotov, Scott A. Smolka and Steve Vickers have corresponded with us about this text; their comments are appreciated. Matt Dwyer and John Hatcliff made useful comments on drafts of Chapter 3. Kevin Lucas provided insightful comments on the content of Chapter 6, and notified us of numerous typographical errors in several drafts of the book. Achim Jung read several chapters and gave useful feedback.

Additionally, a number of people read and provided useful comments on several chapters, including Moti Ben-Ari, Graham Clark, Christian Haack, Anthony Hook, Roberto Segala, Alan Sexton and Allen Stoughton. Numerous students at Kansas State University and the University of Birmingham have given us feedback of various kinds, which has influenced our choice and presentation of the topics. We acknowledge Paul Taylor's LaTeX package for proof boxes. About half a dozen anonymous referees made critical, but constructive, comments which helped to improve this text in various ways. In spite of these contributions, there may still be errors in the book, and we alone must take responsibility for those.

Added for second edition

Many people have helped improve this text by pointing out typos and making other useful comments after the publication date. Among them,

we mention Wolfgang Ahrendt, Yasuhiro Ajiro, Torben Amtoft, Stephan Andrei, Bernhard Beckert, Jonathan Brown, James Caldwell, Ruchira Datta, Amy Felty, Dimitar Guelev, Hirotsugu Kakugawa, Kamran Kashef, Markus Krötzsch, Jagun Kwon, Ranko Lazic, David Makinson, Alexander Miczo, Aart Middeldorp, Robert Morelli, Prakash Panangaden, Aileen Paraguya, Frank Pfenning, Shekhar Pradhan, Koichi Takahashi, Kazunori Ueda, Hiroshi Watanabe, Fuzhi Wang and Reinhard Wilhelm.

1
Propositional logic

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally. Reasoning about situations means constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine.

Consider the following argument:

Example 1.1 If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. *Therefore*, there were taxis at the station.

Intuitively, the argument is valid, since if we put the *first* sentence and the *third* sentence together, they tell us that if there are no taxis, then John will be late. The second sentence tells us that he was not late, so it must be the case that there were taxis.

Much of this book will be concerned with arguments that have this structure, namely, that consist of a number of sentences followed by the word 'therefore' and then another sentence. The argument is valid if the sentence after the 'therefore' logically follows from the sentences before it. Exactly what we mean by 'follows from' is the subject of this chapter and the next one.

Consider another example:

Example 1.2 If it is raining and Jane does not have her umbrella with her, then she will get wet. Jane is not wet. It is raining. *Therefore*, Jane has her umbrella with her.

This is also a valid argument. Closer examination reveals that it actually has the same structure as the argument of the previous example! All we have

done is substituted some sentence fragments for others:

Example 1.1	Example 1.2
the train is late	it is raining
there are taxis at the station	Jane has her umbrella with her
John is late for his meeting	Jane gets wet.

The argument in each example could be stated without talking about trains and rain, as follows:

If p and not q, then r. Not r. p. *Therefore*, q.

In developing logics, we are not concerned with what the sentences really mean, but only in their logical structure. Of course, when we *apply* such reasoning, as done above, such meaning will be of great interest.

1.1 Declarative sentences

In order to make arguments rigorous, we need to develop a language in which we can express sentences in such a way that brings out their logical structure. The language we begin with is the language of propositional logic. It is based on *propositions*, or *declarative sentences* which one can, in principle, argue as being true or false. Examples of declarative sentences are:

(1) The sum of the numbers 3 and 5 equals 8.
(2) Jane reacted violently to Jack's accusations.
(3) Every even natural number >2 is the sum of two prime numbers.
(4) All Martians like pepperoni on their pizza.
(5) Albert Camus était un écrivain français.
(6) Die Würde des Menschen ist unantastbar.

These sentences are all declarative, because they are in principle capable of being declared 'true', or 'false'. Sentence (1) can be tested by appealing to basic facts about arithmetic (and by tacitly assuming an Arabic, decimal representation of natural numbers). Sentence (2) is a bit more problematic. In order to give it a truth value, we need to know who Jane and Jack are and perhaps to have a reliable account from someone who witnessed the situation described. In principle, e.g., if we had been at the scene, we feel that we would have been able to detect Jane's *violent* reaction, provided that it indeed occurred in that way. Sentence (3), known as Goldbach's conjecture, seems straightforward on the face of it. Clearly, a fact about *all* even numbers >2 is either true or false. But to this day nobody knows whether sentence (3) expresses a truth or not. It is even not clear whether this could be shown by some finite means, even if it were true. However, in

this text we will be content with sentences as soon as they can, in principle, attain some truth value regardless of whether this truth value reflects the actual state of affairs suggested by the sentence in question. Sentence (4) seems a bit silly, although we could say that *if* Martians exist and eat pizza, then all of them will either like pepperoni on it or not. (We have to introduce predicate logic in Chapter 2 to see that this sentence is also declarative if *no* Martians exist; it is then true.) Again, for the purposes of this text sentence (4) will do. Et alors, qu'est-ce qu'on pense des phrases (5) et (6)? Sentences (5) and (6) are fine if you happen to read French and German a bit. Thus, declarative statements can be made in any natural, or artificial, language.

The kind of sentences we *won't* consider here are non-declarative ones, like

- Could you please pass me the salt?
- Ready, steady, go!
- May fortune come your way.

Primarily, we are interested in precise declarative sentences, or *statements* about the behaviour of computer systems, or programs. Not only do we want to specify such statements but we also want to *check* whether a given program, or system, fulfils a specification at hand. Thus, we need to develop a calculus of reasoning which allows us to draw conclusions from given assumptions, like initialised variables, which are reliable in the sense that they preserve truth: if all our assumptions are true, then our conclusion ought to be true as well. A much more difficult question is whether, given any true property of a computer program, we can find an argument in our calculus that has this property as its conclusion. The declarative sentence (3) above might illuminate the problematic aspect of such questions in the context of number theory.

The logics we intend to design are *symbolic* in nature. We translate a certain sufficiently large subset of all English declarative sentences into strings of symbols. This gives us a compressed but still complete encoding of declarative sentences and allows us to concentrate on the mere mechanics of our argumentation. This is important since specifications of systems or software are sequences of such declarative sentences. It further opens up the possibility of automatic manipulation of such specifications, a job that computers just love to do[1]. Our strategy is to consider certain declarative sentences as

[1] There is a certain, slightly bitter, circularity in such endeavours: in proving that a certain computer program `P` satisfies a given property, we might let some other computer program `Q` try to find a proof that `P` satisfies the property; but who guarantees us that `Q` satisfies the property of producing only correct proofs? We seem to run into an infinite regress.

being *atomic*, or *indecomposable*, like the sentence

'The number 5 is even.'

We assign certain distinct symbols $p, q, r, \ldots$, or sometimes $p_1, p_2, p_3, \ldots$ to each of these atomic sentences and we can then code up more complex sentences in a *compositional* way. For example, given the atomic sentences

p: 'I won the lottery last week.'
q: 'I purchased a lottery ticket.'
r: 'I won last week's sweepstakes.'

we can form more complex sentences according to the rules below:

$\neg$: The *negation* of p is denoted by $\neg p$ and expresses 'I did **not** win the lottery last week,' or equivalently 'It is **not** true that I won the lottery last week.'

$\vee$: Given p and r we may wish to state that *at least one of them* is true: 'I won the lottery last week, **or** I won last week's sweepstakes;' we denote this declarative sentence by $p \vee r$ and call it the *disjunction* of p and r[2].

$\wedge$: Dually, the formula $p \wedge r$ denotes the rather fortunate *conjunction* of p and r: 'Last week I won the lottery **and** the sweepstakes.'

$\rightarrow$: Last, but definitely not least, the sentence '**If** I won the lottery last week, **then** I purchased a lottery ticket.' expresses an *implication* between p and q, suggesting that q is a logical consequence of p. We write $p \rightarrow q$ for that[3]. We call p the *assumption* of $p \rightarrow q$ and q its *conclusion*.

Of course, we are entitled to use these rules of constructing propositions repeatedly. For example, we are now in a position to form the proposition

$$p \wedge q \rightarrow \neg r \vee q$$

which means that '**if** p **and** q **then not** r **or** q'. You might have noticed a potential ambiguity in this reading. One could have argued that this sentence has the structure 'p is the case **and if** q **then** ...' A computer would require the insertion of brackets, as in

$$(p \wedge q) \rightarrow ((\neg r) \vee q)$$

[2] Its meaning should not be confused with the often implicit meaning of **or** in natural language discourse as **either** ... **or**. In this text **or** always means *at least one of them* and should not be confounded with *exclusive or* which states that *exactly one* of the two statements holds.

[3] The natural language meaning of '**if** ... **then** ...' often implicitly assumes a *causal role* of the assumption somehow enabling its conclusion. The logical meaning of implication is a bit different, though, in the sense that it states the *preservation of truth* which might happen without any causal relationship. For example, 'If all birds can fly, then Bob Dole was never president of the United States of America.' is a true statement, but there is no known causal connection between the flying skills of penguins and effective campaigning.

to disambiguate this assertion. However, we humans get annoyed by a proliferation of such brackets which is why we adopt certain conventions about the *binding priorities* of these symbols.

Convention 1.3 $\neg$ binds more tightly than $\vee$ and $\wedge$, and the latter two bind more tightly than $\rightarrow$. Implication $\rightarrow$ is *right-associative*: expressions of the form $p \rightarrow q \rightarrow r$ denote $p \rightarrow (q \rightarrow r)$.

1.2 Natural deduction

How do we go about constructing a calculus for reasoning about propositions, so that we can establish the validity of Examples 1.1 and 1.2? Clearly, we would like to have a set of rules each of which allows us to draw a conclusion given a certain arrangement of premises.

In natural deduction, we have such a collection of *proof rules*. They allow us to *infer* formulas from other formulas. By applying these rules in succession, we may infer a conclusion from a set of premises.

Let's see how this works. Suppose we have a set of formulas[4] ϕ_1, ϕ_2, ϕ_3, $\ldots$, ϕ_n, which we will call *premises*, and another formula, ψ, which we will call a *conclusion*. By applying proof rules to the premises, we hope to get some more formulas, and by applying more proof rules to those, to eventually obtain the conclusion. This intention we denote by

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi.$$

This expression is called a *sequent*; it is *valid* if a proof for it can be found. The sequent for Examples 1.1 and 1.2 is $p \wedge \neg q \rightarrow r, \neg r, p \vdash q$. Constructing such a proof is a creative exercise, a bit like programming. It is not necessarily obvious which rules to apply, and in what order, to obtain the desired conclusion. Additionally, our proof rules should be carefully chosen; otherwise, we might be able to 'prove' invalid patterns of argumentation. For

[4] It is traditional in logic to use Greek letters. Lower-case letters are used to stand for formulas and upper-case letters are used for sets of formulas. Here are some of the more commonly used Greek letters, together with their pronunciation:

Lower-case		Upper-case	
ϕ	phi	Φ	Phi
ψ	psi	Ψ	Psi
χ	chi	Γ	Gamma
η	eta	Δ	Delta
α	alpha		
β	beta		
γ	gamma		

example, we expect that we won't be able to show the sequent $p, q \vdash p \wedge \neg q$. For example, if p stands for 'Gold is a metal.' and q for 'Silver is a metal,' then knowing these two facts should not allow us to infer that 'Gold is a metal whereas silver isn't.'

Let's now look at our proof rules. We present about fifteen of them in total; we will go through them in turn and then summarise at the end of this section.

1.2.1 Rules for natural deduction

The rules for conjunction Our first rule is called the rule for conjunction ($\wedge$): and-introduction. It allows us to conclude $\phi \wedge \psi$, given that we have already concluded ϕ and ψ separately. We write this rule as

$$\frac{\phi \qquad \psi}{\phi \wedge \psi} \wedge\text{i}.$$

Above the line are the two premises of the rule. Below the line goes the conclusion. (It might not yet be the final conclusion of our argument; we might have to apply more rules to get there.) To the right of the line, we write the name of the rule; $\wedge$i is read 'and-introduction'. Notice that we have introduced a $\wedge$ (in the conclusion) where there was none before (in the premises).

For each of the connectives, there is one or more rules to introduce it and one or more rules to eliminate it. The rules for and-elimination are these two:

$$\frac{\phi \wedge \psi}{\phi} \wedge\text{e}_1 \qquad \frac{\phi \wedge \psi}{\psi} \wedge\text{e}_2. \tag{1.1}$$

The rule $\wedge\text{e}_1$ says: if you have a proof of $\phi \wedge \psi$, then by applying this rule you can get a proof of ϕ. The rule $\wedge\text{e}_2$ says the same thing, but allows you to conclude ψ instead. Observe the dependences of these rules: in the first rule of (1.1), the conclusion ϕ has to match the first conjunct of the premise, whereas the exact nature of the second conjunct ψ is irrelevant. In the second rule it is just the other way around: the conclusion ψ has to match the second conjunct ψ and ϕ can be any formula. It is important to engage in this kind of *pattern matching* before the application of proof rules.

Example 1.4 Let's use these rules to prove that $p \wedge q, r \vdash q \wedge r$ is valid. We start by writing down the premises; then we leave a gap and write the

conclusion:

$$p \wedge q$$
$$r$$

$$q \wedge r$$

The task of constructing the proof is to fill the gap between the premises and the conclusion by applying a suitable sequence of proof rules. In this case, we apply $\wedge e_2$ to the first premise, giving us q. Then we apply $\wedge i$ to this q and to the second premise, r, giving us $q \wedge r$. That's it! We also usually number all the lines, and write in the justification for each line, producing this:

1	$p \wedge q$	premise
2	r	premise
3	q	$\wedge e_2$ 1
4	$q \wedge r$	$\wedge i$ 3, 2

Demonstrate to yourself that you've understood this by trying to show on your own that $(p \wedge q) \wedge r,\ s \wedge t \vdash q \wedge s$ is valid. Notice that the ϕ and ψ can be instantiated not just to atomic sentences, like p and q in the example we just gave, but also to compound sentences. Thus, from $(p \wedge q) \wedge r$ we can deduce $p \wedge q$ by applying $\wedge e_1$, instantiating ϕ to $p \wedge q$ and ψ to r.

If we applied these proof rules literally, then the proof above would actually be a tree with root $q \wedge r$ and leaves $p \wedge q$ and r, like this:

$$\dfrac{\dfrac{p \wedge q}{q}\,\wedge e_2 \qquad r}{q \wedge r}\,\wedge i$$

However, we flattened this tree into a linear presentation which necessitates the use of pointers as seen in lines 3 and 4 above. These pointers allow us to recreate the actual proof tree. Throughout this text, we will use the flattened version of presenting proofs. That way you have to concentrate only on finding a proof, not on how to fit a growing tree onto a sheet of paper.

If a sequent is valid, there may be many different ways of proving it. So if you compare your solution to these exercises with those of others, they need not coincide. The important thing to realise, though, is that any putative proof can be *checked* for correctness by checking each individual line, starting at the top, for the valid application of its proof rule.

The rules of double negation Intuitively, there is no difference between a formula ϕ and its *double negation* $\neg\neg\phi$, which expresses no more and nothing less than ϕ itself. The sentence

'It is **not** true that it does **not** rain.'

is just a more contrived way of saying

'It rains.'

Conversely, knowing 'It rains,' we are free to state this fact in this more complicated manner if we wish. Thus, we obtain rules of elimination and introduction for double negation:

$$\frac{\neg\neg\phi}{\phi}\,\neg\neg\text{e} \qquad \frac{\phi}{\neg\neg\phi}\,\neg\neg\text{i}.$$

(There are rules for single negation on its own, too, which we will see later.)

Example 1.5 The proof of the sequent $p, \neg\neg(q \wedge r) \vdash \neg\neg p \wedge r$ below uses most of the proof rules discussed so far:

1	p	premise
2	$\neg\neg(q \wedge r)$	premise
3	$\neg\neg p$	$\neg\neg$i 1
4	$q \wedge r$	$\neg\neg$e 2
5	r	$\wedge\text{e}_2$ 4
6	$\neg\neg p \wedge r$	$\wedge$i 3, 5

Example 1.6 We now prove the sequent $(p \wedge q) \wedge r,\ s \wedge t \vdash q \wedge s$ which you were invited to prove by yourself in the last section. Please compare the proof below with your solution:

1	$(p \wedge q) \wedge r$	premise
2	$s \wedge t$	premise
3	$p \wedge q$	$\wedge\text{e}_1$ 1
4	q	$\wedge\text{e}_2$ 3
5	s	$\wedge\text{e}_1$ 2
6	$q \wedge s$	$\wedge$i 4, 5

The rule for eliminating implication There is one rule to introduce $\rightarrow$ and one to eliminate it. The latter is one of the best known rules of propositional logic and is often referred to by its Latin name *modus ponens*. We will usually call it by its modern name, implies-elimination (sometimes also referred to as arrow-elimination). This rule states that, given ϕ and knowing that ϕ implies ψ, we may rightfully conclude ψ. In our calculus, we write this as

$$\frac{\phi \quad \phi \rightarrow \psi}{\psi} \rightarrow\text{e}.$$

Let us justify this rule by spelling out instances of some declarative sentences p and q. Suppose that

$$\begin{aligned} p &: \text{It rained.} \\ p \rightarrow q &: \text{If it rained, then the street is wet.} \end{aligned}$$

so q is just 'The street is wet.' Now, *if* we know that it rained and *if* we know that the street is wet in the case that it rained, then we may combine these two pieces of information to conclude that the street is indeed wet. Thus, the justification of the $\rightarrow$e rule is a mere application of common sense. Another example from programming is:

$$\begin{aligned} p &: \text{The value of the program's input is an integer.} \\ p \rightarrow q &: \text{If the program's input is an integer, then the program outputs a boolean.} \end{aligned}$$

Again, we may put all this together to conclude that our program outputs a boolean value if supplied with an integer input. However, it is important to realise that the presence of p is absolutely essential for the inference to happen. For example, our program might well satisfy $p \rightarrow q$, but if it doesn't satisfy p – e.g. if its input is a surname – then we will not be able to derive q.

As we saw before, the formal parameters ϕ and the ψ for $\rightarrow$e can be instantiated to any sentence, including compound ones:

1	$\neg p \wedge q$	premise
2	$\neg p \wedge q \rightarrow r \vee \neg p$	premise
3	$r \vee \neg p$	$\rightarrow$e 2, 1

Of course, we may use any of these rules as often as we wish. For example, given p, $p \to q$ and $p \to (q \to r)$, we may infer r:

1	$p \to (q \to r)$	premise
2	$p \to q$	premise
3	p	premise
4	$q \to r$	$\to$e 1, 3
5	q	$\to$e 2, 3
6	r	$\to$e 4, 5

Before turning to implies-introduction, let's look at a hybrid rule which has the Latin name *modus tollens*. It is like the $\to$e rule in that it eliminates an implication. Suppose that $p \to q$ *and* $\neg q$ are the case. Then, *if* p holds we can use $\to$e to conclude that q holds. Thus, we then have that q *and* $\neg q$ hold, which is impossible. Therefore, we may infer that p must be false. But this can only mean that $\neg p$ is true. We summarise this reasoning into the rule *modus tollens*, or MT for short:[5]

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ \text{MT}.$$

Again, let us see an example of this rule in the natural language setting:

> *'If Abraham Lincoln was Ethiopian, then he was African. Abraham Lincoln was not African; therefore he was not Ethiopian.'*

Example 1.7 In the following proof of

$$p \to (q \to r),\ p,\ \neg r \vdash \neg q$$

we use several of the rules introduced so far:

1	$p \to (q \to r)$	premise
2	p	premise
3	$\neg r$	premise
4	$q \to r$	$\to$e 1, 2
5	$\neg q$	MT 4, 3

[5] We will be able to *derive* this rule from other ones later on, but we introduce it here because it allows us already to do some pretty slick proofs. You may think of this rule as one on a higher level insofar as it does not mention the lower-level rules upon which it depends.

Examples 1.8 Here are two example proofs which combine the rule MT with either ¬¬e or ¬¬i:

1	$\neg p \to q$	premise
2	$\neg q$	premise
3	$\neg\neg p$	MT 1, 2
4	p	¬¬e 3

proves that the sequent $\neg p \to q,\ \neg q \vdash p$ is valid; and

1	$p \to \neg q$	premise
2	q	premise
3	$\neg\neg q$	¬¬i 2
4	$\neg p$	MT 1, 3

shows the validity of the sequent $p \to \neg q,\ q \vdash \neg p$.

Note that the order of applying double negation rules and MT is different in these examples; this order is driven by the structure of the particular sequent whose validity one is trying to show.

The rule implies introduction The rule MT made it possible for us to show that $p \to q,\ \neg q \vdash \neg p$ is valid. But the validity of the sequent $p \to q \vdash \neg q \to \neg p$ seems just as plausible. That sequent is, in a certain sense, saying the same thing. Yet, so far we have no rule which *builds* implications that do not already occur as premises in our proofs. The mechanics of such a rule are more involved than what we have seen so far. So let us proceed with care. Let us suppose that $p \to q$ is the case. If we *temporarily* assume that $\neg q$ holds, we can use MT to infer $\neg p$. Thus, assuming $p \to q$ we can show that $\neg q$ **implies** $\neg p$; but the latter we express *symbolically* as $\neg q \to \neg p$. To summarise, we have found an argumentation for $p \to q \vdash \neg q \to \neg p$:

1	$p \to q$	premise
2	$\neg q$	assumption
3	$\neg p$	MT 1, 2
4	$\neg q \to \neg p$	→i 2–3

The box in this proof serves to demarcate the scope of the temporary assumption $\neg q$. What we are saying is: let's make the assumption of $\neg q$. To

do this, we open a box and put $\neg q$ at the top. Then we continue applying other rules as normal, for example to obtain $\neg p$. But this still depends on the assumption of $\neg q$, so it goes inside the box. Finally, we are ready to apply $\rightarrow$i. It allows us to conclude $\neg q \rightarrow \neg p$, but that conclusion no longer *depends* on the assumption $\neg q$. Compare this with saying that 'If you are French, then you are European.' The truth of this sentence does not depend on whether anybody is French or not. Therefore, we write the conclusion $\neg q \rightarrow \neg p$ outside the box.

This works also as one would expect if we think of $p \rightarrow q$ as a *type* of a procedure. For example, p could say that the procedure expects an integer value x as input and q might say that the procedure returns a boolean value y as output. The validity of $p \rightarrow q$ amounts now to an assume-guarantee assertion: if the input is an integer, then the output is a boolean. This assertion can be true about a procedure while that same procedure could compute strange things or crash in the case that the input is not an integer. Showing $p \rightarrow q$ using the rule $\rightarrow$i is now called *type checking*, an important topic in the construction of compilers for typed programming languages.

We thus formulate the rule $\rightarrow$i as follows:

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \rightarrow \psi}\ \rightarrow\text{i}.$$

It says: in order to prove $\phi \rightarrow \psi$, make a temporary assumption of ϕ and then prove ψ. In your proof of ψ, you can use ϕ and any of the other formulas such as premises and provisional conclusions that you have made so far. Proofs may nest boxes or open new boxes after old ones have been closed. There are rules about which formulas can be used at which points in the proof. Generally, we can only use a formula ϕ in a proof at a given point if that formula occurs *prior* to that point and if no box which encloses that occurrence of ϕ has been closed already.

The line immediately following a closed box has to match the pattern of the conclusion of the rule that uses the box. For implies-introduction, this means that we have to continue after the box with $\phi \rightarrow \psi$, where ϕ was the first and ψ the last formula of that box. We will encounter two more proof rules involving proof boxes and they will require similar pattern matching.

Example 1.9 Here is another example of a proof using →i:

1	$\neg q \to \neg p$	premise
2	p	assumption
3	$\neg\neg p$	¬¬i 2
4	$\neg\neg q$	MT 1, 3
5	$p \to \neg\neg q$	→i 2–4

which verifies the validity of the sequent $\neg q \to \neg p \vdash p \to \neg\neg q$. Notice that we could apply the rule MT to formulas occurring in or above the box: at line 4, no box has been closed that would enclose line 1 or 3.

At this point it is instructive to consider the one-line argument

1	p	premise

which demonstrates $p \vdash p$. The rule →i (with conclusion $\phi \to \psi$) does not prohibit the possibility that ϕ and ψ coincide. They could both be instantiated to p. Therefore we may extend the proof above to

1	p	assumption
2	$p \to p$	→i 1 – 1

We write $\vdash p \to p$ to express that the argumentation for $p \to p$ does not depend on any premises at all.

Definition 1.10 Logical formulas ϕ with valid sequent $\vdash \phi$ are *theorems*.

Example 1.11 Here is an example of a theorem whose proof utilises most of the rules introduced so far:

1	$q \to r$	assumption
2	$\neg q \to \neg p$	assumption
3	p	assumption
4	$\neg\neg p$	¬¬i 3
5	$\neg\neg q$	MT 2, 4
6	q	¬¬e 5
7	r	→e 1, 6
8	$p \to r$	→i 3–7
9	$(\neg q \to \neg p) \to (p \to r)$	→i 2–8
10	$(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$	→i 1–9

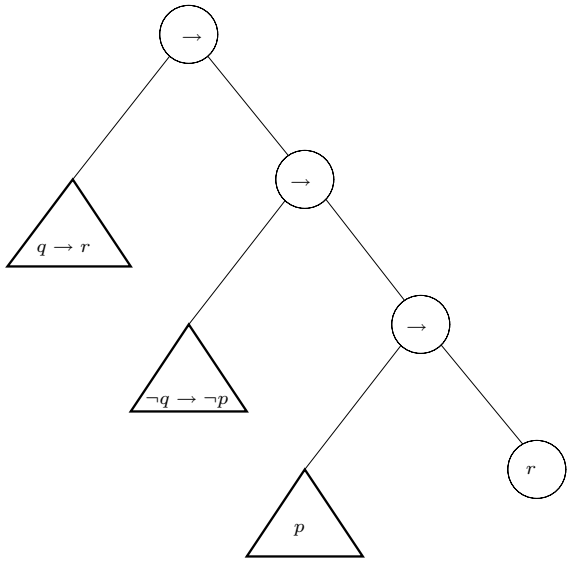


Figure 1.1. Part of the structure of the formula $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ to show how it determines the proof structure.

Therefore the sequent $\vdash (q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ is valid, showing that $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ is another theorem.

Remark 1.12 Indeed, this example indicates that we may transform any proof of $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ in such a way into a proof of the theorem

$$\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\cdots \to (\phi_n \to \psi) \ldots)))$$

by 'augmenting' the previous proof with n lines of the rule $\to$i applied to ϕ_n, ϕ_{n-1},..., ϕ_1 in that order.

The nested boxes in the proof of Example 1.11 reveal a pattern of using elimination rules first, to deconstruct assumptions we have made, and then introduction rules to construct our final conclusion. More difficult proofs may involve several such phases.

Let us dwell on this important topic for a while. How did we come up with the proof above? Parts of it are *determined* by the structure of the formulas we have, while other parts require us to be *creative*. Consider the logical structure of $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ schematically depicted in Figure 1.1. The formula is overall an implication since $\to$ is the root of the tree in Figure 1.1. But the only way to build an implication is by means

of the rule →i. Thus, we need to state the assumption of that implication as such (line 1) and have to show its conclusion (line 9). If we managed to do that, then we know how to end the proof in line 10. In fact, as we already remarked, this is the only way we could have ended it. So essentially lines 1, 9 and 10 are completely determined by the structure of the formula; further, we have reduced the problem to filling the gaps in between lines 1 and 9. But again, the formula in line 9 is an implication, so we have only one way of showing it: assuming its premise in line 2 and trying to show its conclusion in line 8; as before, line 9 is obtained by →i. The formula $p \to r$ in line 8 is yet another implication. Therefore, we have to assume p in line 3 and hope to show r in line 7, then →i produces the desired result in line 8.

The remaining question now is this: how can we show r, using the three assumptions in lines 1–3? This, and only this, is the creative part of this proof. We see the implication $q \to r$ in line 1 and know how to get r (using →e) if only we had q. So how could we get q? Well, lines 2 and 3 almost look like a pattern for the MT rule, which would give us $\neg\neg q$ in line 5; the latter is quickly changed to q in line 6 via ¬¬e. However, the pattern for MT does not match right away, since it requires $\neg\neg p$ instead of p. But this is easily accomplished via ¬¬i in line 4.

The moral of this discussion is that the logical structure of the formula to be shown tells you a lot about the structure of a possible proof and it is definitely worth your while to exploit that information in trying to prove sequents. Before ending this section on the rules for implication, let's look at some more examples (this time also involving the rules for conjunction).

Example 1.13 Using the rule ∧i, we can prove the validity of the sequent $p \wedge q \to r \vdash p \to (q \to r)$:

1	$p \wedge q \to r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	∧i 2, 3
5	r	→e 1, 4
6	$q \to r$	→i 3–5
7	$p \to (q \to r)$	→i 2–6