Michael Huth and Mark Ryan

Second Edition

Logic in Computer Science

Modelling and Reasoning about Systems

CAMBRIDGE

CAMBRIDGE

www.cambridge.org/9780521543101

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

Cambridge University Press has no responsibility for the persistence or accuracy of urls for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

Foreword to the first edition *page* ix
Preface to the second edition xi
Acknowledgements xiii
1 Propositional logic 1
1.1 Declarative sentences 2
1.2 Natural deduction 5
1.2.1 Rules for natural deduction 6
1.2.2 Derived rules 23
1.2.3 Natural deduction in summary 26
1.2.4 Provable equivalence 29
1.2.5 An aside: proof by contradiction 29
1.3 Propositional logic as a formal language 31
1.4 Semantics of propositional logic 36
1.4.1 The meaning of logical connectives 36
1.4.2 Mathematical induction 40
1.4.3 Soundness of propositional logic 45
1.4.4 Completeness of propositional logic 49
1.5 Normal forms 53
1.5.1 Semantic equivalence, satisfiability and validity 54
1.5.2 Conjunctive normal forms and validity 58
1.5.3 Horn clauses and satisfiability 65
1.6 SAT solvers 68
1.6.1 A linear solver 69
1.6.2 A cubic solver 72
1.7 Exercises 78
1.8 Bibliographic notes 91
2 Predicate logic 93
2.1 The need for a richer language 93

2.2 Predicate logic as a formal language 98
2.2.1 Terms 99
2.2.2 Formulas 100
2.2.3 Free and bound variables 102
2.2.4 Substitution 104
2.3 Proof theory of predicate logic 107
2.3.1 Natural deduction rules 107
2.3.2 Quantifier equivalences 117
2.4 Semantics of predicate logic 122
2.4.1 Models 123
2.4.2 Semantic entailment 129
2.4.3 The semantics of equality 130
2.5 Undecidability of predicate logic 131
2.6 Expressiveness of predicate logic 136
2.6.1 Existential second-order logic 139
2.6.2 Universal second-order logic 140
2.7 Micromodels of software 141
2.7.1 State machines 142
2.7.2 Alma – re-visited 146
2.7.3 A software micromodel 148
2.8 Exercises 157
2.9 Bibliographic notes 170
3 Verification by model checking 172
3.1 Motivation for verification 172
3.2 Linear-time temporal logic 175
3.2.1 Syntax of LTL 175
3.2.2 Semantics of LTL 178
3.2.3 Practical patterns of specifications 183
3.2.4 Important equivalences between LTL formulas 184
3.2.5 Adequate sets of connectives for LTL 186
3.3 Model checking: systems, tools, properties 187
3.3.1 Example: mutual exclusion 187
3.3.2 The NuSMV model checker 191
3.3.3 Running NuSMV 194
3.3.4 Mutual exclusion revisited 195
3.3.5 The ferryman 199
3.3.6 The alternating bit protocol 203
3.4 Branching-time logic 207
3.4.1 Syntax of CTL 208

3.4.2 Semantics of CTL 211
3.4.3 Practical patterns of specifications 215
3.4.4 Important equivalences between CTL formulas 215
3.4.5 Adequate sets of CTL connectives 216
3.5 CTL* and the expressive powers of LTL and CTL 217
3.5.1 Boolean combinations of temporal formulas in CTL 220
3.5.2 Past operators in LTL 221
3.6 Model-checking algorithms 221
3.6.1 The CTL model-checking algorithm 222
3.6.2 CTL model checking with fairness 230
3.6.3 The LTL model-checking algorithm 232
3.7 The fixed-point characterisation of CTL 238
3.7.1 Monotone functions 240
3.7.2 The correctness of $\mathtt{SAT}_{\mathtt{EG}}$ 242
3.7.3 The correctness of $\mathtt{SAT}_{\mathtt{EU}}$ 243
3.8 Exercises 245
3.9 Bibliographic notes 254
4 Program verification 256
4.1 Why should we specify and verify code? 257
4.2 A framework for software verification 258
4.2.1 A core programming language 259
4.2.2 Hoare triples 262
4.2.3 Partial and total correctness 265
4.2.4 Program variables and logical variables 268
4.3 Proof calculus for partial correctness 269
4.3.1 Proof rules 269
4.3.2 Proof tableaux 273
4.3.3 A case study: minimal-sum section 287
4.4 Proof calculus for total correctness 292
4.5 Programming by contract 296
4.6 Exercises 299
4.7 Bibliographic notes 304
5 Modal logics and agents 306
5.1 Modes of truth 306
5.2 Basic modal logic 307
5.2.1 Syntax 307
5.2.2 Semantics 308
5.3 Logic engineering 316
5.3.1 The stock of valid formulas 317

5.3.2 Important properties of the accessibility relation 320
5.3.3 Correspondence theory 322
5.3.4 Some modal logics 326
5.4 Natural deduction 328
5.5 Reasoning about knowledge in a multi-agent system 331
5.5.1 Some examples 332
5.5.2 The modal logic $\mathrm{KT45}^n$ 335
5.5.3 Natural deduction for $\mathrm{KT45}^n$ 339
5.5.4 Formalising the examples 342
5.6 Exercises 350
5.7 Bibliographic notes 356
6 Binary decision diagrams 358
6.1 Representing boolean functions 358
6.1.1 Propositional formulas and truth tables 359
6.1.2 Binary decision diagrams 361
6.1.3 Ordered BDDs 366
6.2 Algorithms for reduced OBDDs 372
6.2.1 The algorithm `reduce` 372
6.2.2 The algorithm `apply` 373
6.2.3 The algorithm `restrict` 377
6.2.4 The algorithm `exists` 377
6.2.5 Assessment of OBDDs 380
6.3 Symbolic model checking 382
6.3.1 Representing subsets of the set of states 383
6.3.2 Representing the transition relation 385
6.3.3 Implementing the functions $\mathrm{pre}_\exists$ and $\mathrm{pre}_\forall$ 387
6.3.4 Synthesising OBDDs 387
6.4 A relational mu-calculus 390
6.4.1 Syntax and semantics 390
6.4.2 Coding CTL models and specifications 393
6.5 Exercises 398
6.6 Bibliographic notes 413
Bibliography 414
Index 418

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*

$$\mathcal{M} \vDash \phi$$

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 $\vDash$. 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.

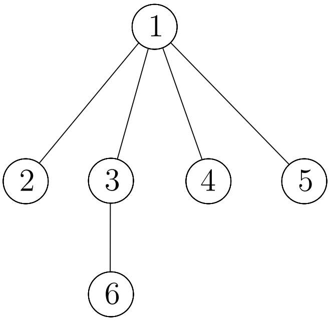
Chapter 3 has been completely restructured. It now begins with a discussion of linear-time temporal logic; features the open-source NuSMV model-checking tool throughout; and includes a discussion on planning problems, more material on the expressiveness of temporal logics, and new modelling examples.

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, …, 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

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

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:

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

The rule $\wedge\mathrm{e}_1$ says: if you have a proof of $\phi \wedge \psi$, then by applying this rule you can get a proof of ϕ. The rule $\wedge\mathrm{e}_2$ 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:

$$\begin{array}{l} p \wedge q \\ r \\ \\ q \wedge r \end{array}$$

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:

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

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:

$$\frac{\neg\neg\phi}{\phi}\ \neg\neg\mathrm{e} \qquad\qquad \frac{\phi}{\neg\neg\phi}\ \neg\neg\mathrm{i}.$$

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

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

1	p	premise
2	$\neg\neg(q \land r)$	premise
3	$\neg\neg p$	$\neg\neg$i 1
4	$q \land r$	$\neg\neg$e 2
5	r	$\land$e$_2$ 4
6	$\neg\neg p \land r$	$\land$i 3, 5

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

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

The rule for eliminating implication There is one rule to introduce $\to$ 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

$$\frac{\phi \quad \phi \to \psi}{\psi} \to\mathrm{e}.$$

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

p: It rained.
$p \to q$: If it rained, then the street is wet.

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 $\to$e rule is a mere application of common sense. Another example from programming is:

p: The value of the program's input is an integer.
$p \to q$: If the program's input is an integer, then the program outputs a boolean.

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 $p \to q$, 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 $\to$e can be instantiated to any sentence, including compound ones:

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

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

1	$p \rightarrow (q \rightarrow r)$	premise
2	$p \rightarrow q$	premise
3	p	premise
4	$q \rightarrow r$	$\rightarrow$e $1, 3$
5	q	$\rightarrow$e $2, 3$
6	r	$\rightarrow$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 →e rule in that it eliminates an implication. Suppose that $p \rightarrow q$ *and* $\neg q$ are the case. Then, *if* p holds we can use →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]

$$\frac{\phi \rightarrow \psi \quad \neg\psi}{\neg\phi}\ \mathrm{MT}.$$

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

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

we use several of the rules introduced so far:

1	$p \rightarrow (q \rightarrow r)$	premise
2	p	premise
3	$\neg r$	premise
4	$q \rightarrow r$	$\rightarrow$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 $\to$i. It allows us to conclude $\neg q \to \neg p$, 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 $\neg q \to \neg p$ outside the box.

This works also as one would expect if we think of $p \to q$ 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 $p \to q$ 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 $p \to q$ using the rule $\to$i is now called *type checking*, an important topic in the construction of compilers for typed programming languages.

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

$$\frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \to \psi}\ \to\mathrm{i}.$$

It says: in order to prove $\phi \to \psi$, 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 $\phi \to \psi$, 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	$\neg\neg p$	$\neg\neg$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	$\neg\neg p$	$\neg\neg$i 3
5	$\neg\neg q$	MT 2, 4
6	q	$\neg\neg$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

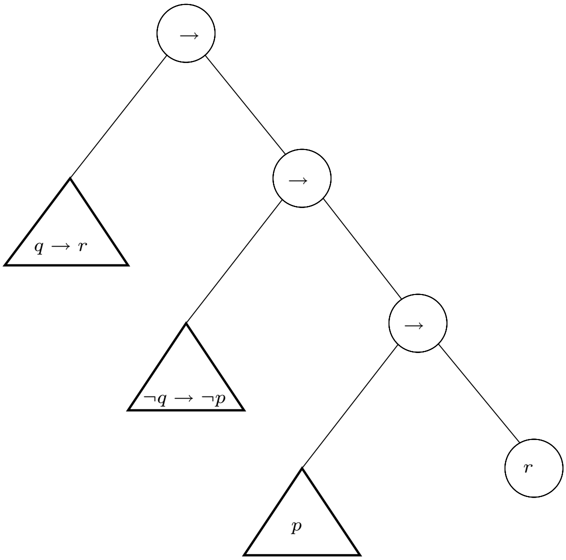


Figure 1.1. Part of the structure of the formula $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ to show how it determines the proof structure.

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, $\phi_{n-1}, \ldots$, 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 $p \rightarrow r$ 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 $q \rightarrow r$ 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

$$p \wedge q \rightarrow r \vdash p \rightarrow (q \rightarrow r):$$

1	$p \wedge q \rightarrow r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	$\wedge$i $2, 3$
5	r	$\rightarrow$e $1, 4$
6	$q \rightarrow r$	$\rightarrow$i $3-5$
7	$p \rightarrow (q \rightarrow r)$	$\rightarrow$i $2-6$

Example 1.14 Using the two elimination rules $\wedge\mathrm{e}_1$ and $\wedge\mathrm{e}_2$, we can show that the 'converse' of the sequent above is valid, too:

1	$p \to (q \to r)$	premise
2	$p \wedge q$	assumption
3	p	$\wedge\mathrm{e}_1$ 2
4	q	$\wedge\mathrm{e}_2$ 2
5	$q \to r$	$\to$e $1, 3$
6	r	$\to$e $5, 4$
7	$p \wedge q \to r$	$\to$i 2–6

The validity of $p \to (q \to r) \vdash p \wedge q \to r$ and $p \wedge q \to r \vdash p \to (q \to r)$ means that these two formulas are equivalent in the sense that we can prove one from the other. We denote this by

$$p \wedge q \to r \dashv\vdash p \to (q \to r).$$

Since there can be only one formula to the right of $\vdash$, we observe that each instance of $\dashv\vdash$ can only relate *two* formulas to each other.

Example 1.15 Here is an example of a proof that uses introduction *and* elimination rules for conjunction; it shows the validity of the sequent $p \to q \vdash p \wedge r \to q \wedge r$:

1	$p \to q$	premise
2	$p \wedge r$	assumption
3	p	$\wedge\mathrm{e}_1$ 2
4	r	$\wedge\mathrm{e}_2$ 2
5	q	$\to$e $1, 3$
6	$q \wedge r$	$\wedge$i $5, 4$
7	$p \wedge r \to q \wedge r$	$\to$i 2–6

The rules for disjunction The rules for disjunction are different in spirit from those for conjunction. The case for conjunction was concise and clear: proofs of $\phi \wedge \psi$ are essentially nothing but a concatenation of a proof of ϕ and a proof of ψ, plus an additional line invoking $\wedge$i. In the case of disjunctions, however, it turns out that the *introduction* of disjunctions is by far easier to grasp than their elimination. So we begin with the rules $\vee\mathrm{i}_1$ and $\vee\mathrm{i}_2$. From the premise ϕ we can infer that 'ϕ **or** ψ' holds, for we already know that ϕ holds. Note that this inference is valid for any choice of ψ. By the same token, we may conclude 'ϕ **or** ψ' if we already have ψ. Similarly, that inference works for any choice of ϕ. Thus, we arrive at the proof rules

$$\frac{\phi}{\phi \vee \psi}\ \vee\mathrm{i}_1 \qquad \frac{\psi}{\phi \vee \psi}\ \vee\mathrm{i}_2.$$

So if p stands for 'Agassi won a gold medal in 1996.' and q denotes the sentence 'Agassi won Wimbledon in 1996.' then $p \vee q$ is the case because p is true, regardless of the fact that q is false. Naturally, the constructed disjunction depends upon the assumptions needed in establishing its respective disjunct p or q.

Now let's consider or-elimination. How can we use a formula of the form $\phi \vee \psi$ in a proof? Again, our guiding principle is to *disassemble* assumptions into their basic constituents so that the latter may be used in our argumentation such that they render our desired conclusion. Let us imagine that we want to show some proposition χ by assuming $\phi \vee \psi$. Since we don't know which of ϕ and ψ is true, we have to give *two* separate proofs which we need to combine into one argument:

1. First, we assume ϕ is true and have to come up with a proof of χ.
2. Next, we assume ψ is true and need to give a proof of χ as well.
3. Given these two proofs, we can infer χ from the truth of $\phi \vee \psi$, since our case analysis above is exhaustive.

Therefore, we write the rule $\vee$e as follows:

$$\frac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ \vee\mathrm{e}.$$

It is saying that: if $\phi \vee \psi$ is true and – no matter whether we assume ϕ or we assume ψ – we can get a proof of χ, then we are entitled to deduce χ anyway. Let's look at a proof that $p \vee q \vdash q \vee p$ is valid:

1	$p \vee q$	premise
2	p	assumption
3	$q \vee p$	$\vee\mathrm{i}_2$ 2
4	q	assumption
5	$q \vee p$	$\vee\mathrm{i}_1$ 4
6	$q \vee p$	$\vee\mathrm{e}$ 1, 2–3, 4–5

Here are some points you need to remember about applying the $\vee$e rule.

- For it to be a sound argument we have to make sure that the conclusions in each of the two cases (the χ in the rule) are actually the same formula.
- The work done by the rule $\vee$e is the combining of the arguments of the two cases into one.
- In each case you may not use the temporary assumption of the other case, unless it is something that has already been shown before those case boxes began.
- The invocation of rule $\vee$e in line 6 lists three things: the line in which the disjunction appears (1), and the location of the two boxes for the two cases (2–3 and 4–5).

If we use $\phi \vee \psi$ in an argument where it occurs only as an assumption or a premise, then we are missing a certain amount of information: we know ϕ, or ψ, but we don't know which one of the two it is. Thus, we have to make a solid case for each of the two possibilities ϕ or ψ; this resembles the behaviour of a `CASE` or `IF` statement found in most programming languages.

Example 1.16 Here is a more complex example illustrating these points. We prove that the sequent $q \to r \vdash p \vee q \to p \vee r$ is valid:

1	$q \to r$	premise
2	$p \vee q$	assumption
3	p	assumption
4	$p \vee r$	$\vee\mathrm{i}_1$ 3
5	q	assumption
6	r	$\to$e $1, 5$
7	$p \vee r$	$\vee\mathrm{i}_2$ 6
8	$p \vee r$	$\vee$e $2, 3{-}4, 5{-}7$
9	$p \vee q \to p \vee r$	$\to$i $2{-}8$

Note that the propositions in lines 4, 7 and 8 coincide, so the application of $\vee$e is legitimate.

We give some more example proofs which use the rules $\vee$e, $\vee\mathrm{i}_1$ and $\vee\mathrm{i}_2$.

Example 1.17 Proving the validity of the sequent $(p \vee q) \vee r \vdash p \vee (q \vee r)$ is surprisingly long and seemingly complex. But this is to be expected, since the elimination rules break $(p \vee q) \vee r$ up into its atomic constituents p, q and r, whereas the introduction rules then built up the formula $p \vee (q \vee r)$.

1	$(p \vee q) \vee r$	premise
2	$(p \vee q)$	assumption
3	p	assumption
4	$p \vee (q \vee r)$	$\vee\mathrm{i}_1$ 3
5	q	assumption
6	$q \vee r$	$\vee\mathrm{i}_1$ 5
7	$p \vee (q \vee r)$	$\vee\mathrm{i}_2$ 6
8	$p \vee (q \vee r)$	$\vee\mathrm{e}$ $2, 3–4, 5–7$
9	r	assumption
10	$q \vee r$	$\vee\mathrm{i}_2$ 9
11	$p \vee (q \vee r)$	$\vee\mathrm{i}_2$ 10
12	$p \vee (q \vee r)$	$\vee\mathrm{e}$ $1, 2–8, 9–11$

Example 1.18 From boolean algebra, or circuit theory, you may know that disjunctions distribute over conjunctions. We are now able to prove this in natural deduction. The following proof:

1	$p \wedge (q \vee r)$	premise
2	p	$\wedge\mathrm{e}_1$ 1
3	$q \vee r$	$\wedge\mathrm{e}_2$ 1
4	q	assumption
5	$p \wedge q$	$\wedge\mathrm{i}$ $2, 4$
6	$(p \wedge q) \vee (p \wedge r)$	$\vee\mathrm{i}_1$ 5
7	r	assumption
8	$p \wedge r$	$\wedge\mathrm{i}$ $2, 7$
9	$(p \wedge q) \vee (p \wedge r)$	$\vee\mathrm{i}_2$ 8
10	$(p \wedge q) \vee (p \wedge r)$	$\vee\mathrm{e}$ $3, 4–6, 7–9$

verifies the validity of the sequent $p \wedge (q \vee r) \vdash (p \wedge q) \vee (p \wedge r)$ and you are encouraged to show the validity of the 'converse' $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$ yourself.

A final rule is required in order to allow us to conclude a box with a formula which has already appeared earlier in the proof. Consider the sequent $\vdash p \to (q \to p)$, whose validity may be proved as follows:

1	p	assumption
2	q	assumption
3	p	copy 1
4	$q \to p$	$\to$i 2–3
5	$p \to (q \to p)$	$\to$i 1–4

The rule ‘copy’ allows us to repeat something that we know already. We need to do this in this example, because the rule $\to$i requires that we end the inner box with p. The copy rule entitles us to copy formulas that appeared before, unless they depend on temporary assumptions whose box has already been closed. Though a little inelegant, this additional rule is a small price to pay for the freedom of being able to use premises, or any other ‘visible’ formulas, more than once.

The rules for negation We have seen the rules $\neg\neg$i and $\neg\neg$e, but we haven’t seen any rules that introduce or eliminate single negations. These rules involve the notion of *contradiction*. This detour is to be expected since our reasoning is concerned about the inference, and therefore the preservation, of truth. Hence, there cannot be a direct way of inferring $\neg\phi$, given ϕ.

Definition 1.19 Contradictions are expressions of the form $\phi \wedge \neg\phi$ or $\neg\phi \wedge \phi$, where ϕ is any formula.

Examples of such contradictions are $r \wedge \neg r$, $(p \to q) \wedge \neg(p \to q)$ and $\neg(r \vee s \to q) \wedge (r \vee s \to q)$. Contradictions are a very important notion in logic. As far as truth is concerned, they are all equivalent; that means we should be able to prove the validity of

$$\neg(r \vee s \to q) \wedge (r \vee s \to q) \dashv\vdash (p \to q) \wedge \neg(p \to q) \tag{1.2}$$

since both sides are contradictions. We’ll be able to prove this later, when we have introduced the rules for negation.

Indeed, it’s not just that contradictions can be derived from contradictions; actually, *any* formula can be derived from a contradiction. This can be confusing when you first encounter it; why should we endorse the argument $p \wedge \neg p \vdash q$, where

$$p: \text{The moon is made of green cheese.}$$
$$q: \text{I like pepperoni on my pizza.}$$

considering that our taste in pizza doesn't have anything to do with the constitution of the moon? On the face of it, such an endorsement may seem absurd. Nevertheless, natural deduction does have this feature that any formula can be derived from a contradiction and therefore it makes this argument valid. The reason it takes this stance is that $\vdash$ tells us all the things we may infer, provided that we can assume the formulas to the left of it. This process does not care whether such premises make any sense. This has at least the advantage that we can match $\vdash$ to checks based on semantic intuitions which we formalise later by using truth tables: if all the premises compute to 'true', then the conclusion must compute 'true' as well. In particular, this is not a constraint in the case that one of the premises is (always) false.

The fact that $\bot$ can prove anything is encoded in our calculus by the proof rule bottom-elimination:

$$\frac{\bot}{\phi}\ \bot\mathrm{e}.$$

The fact that $\bot$ itself represents a contradiction is encoded by the proof rule not-elimination:

$$\frac{\phi \quad \neg\phi}{\bot}\ \neg\mathrm{e}.$$

Example 1.20 We apply these rules to show that $\neg p \vee q \vdash p \rightarrow q$ is valid:

1	$\neg p \vee q$			
2	$\neg p$	premise	q	premise
3	p	assumption	p	assumption
4	$\bot$	$\neg$e 3, 2	q	copy 2
5	q	$\bot$e 4	$p \rightarrow q$	$\rightarrow$i 3–4
6	$p \rightarrow q$	$\rightarrow$i 3–5		
7	$p \rightarrow q$			$\vee$e 1, 2–6

Notice how, in this example, the proof boxes for $\vee$e are drawn side by side instead of on top of each other. It doesn't matter which way you do it.

What about introducing negations? Well, suppose we make an assumption which gets us into a contradictory state of affairs, i.e. gets us $\bot$. Then our assumption cannot be true; so it must be false. This intuition is the basis for the proof rule $\neg$i:

$$\frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}\ \neg\text{i}.$$

Example 1.21 We put these rules in action, demonstrating that the sequent $p \to q,\ p \to \neg q \vdash \neg p$ is valid:

1	$p \to q$	premise
2	$p \to \neg q$	premise
3	p	assumption
4	q	$\to$e $1, 3$
5	$\neg q$	$\to$e $2, 3$
6	$\bot$	$\neg$e $4, 5$
7	$\neg p$	$\neg$i $3-6$

Lines 3–6 contain all the work of the $\neg$i rule. Here is a second example, showing the validity of a sequent, $p \to \neg p \vdash \neg p$, with a contradictory formula as sole premise:

1	$p \to \neg p$	premise
2	p	assumption
3	$\neg p$	$\to$e $1, 2$
4	$\bot$	$\neg$e $2, 3$
5	$\neg p$	$\neg$i $2-4$

Example 1.22 We prove that the sequent $p \to (q \to r),\ p,\ \neg r \vdash \neg q$ is valid,

without using the MT rule:

1	$p \to (q \to r)$	premise
2	p	premise
3	$\neg r$	premise
4	q	assumption
5	$q \to r$	$\to$e $1, 2$
6	r	$\to$e $5, 4$
7	$\bot$	$\neg$e $6, 3$
8	$\neg q$	$\neg$i 4–7

Example 1.23 Finally, we return to the argument of Examples 1.1 and 1.2, which can be coded up by the sequent $p \wedge \neg q \to r$, $\neg r$, $p \vdash q$ whose validity we now prove:

1	$p \wedge \neg q \to r$	premise
2	$\neg r$	premise
3	p	premise
4	$\neg q$	assumption
5	$p \wedge \neg q$	$\wedge$i $3, 4$
6	r	$\to$e $1, 5$
7	$\bot$	$\neg$e $6, 2$
8	$\neg\neg q$	$\neg$i 4–7
9	q	$\neg\neg$e 8

1.2.2 Derived rules

When describing the proof rule *modus tollens* (MT), we mentioned that it is not a primitive rule of natural deduction, but can be derived from some of the other rules. Here is the derivation of

$$\frac{\phi \to \psi \qquad \neg\psi}{\neg\phi}\ \mathrm{MT}$$

from →e, ¬e and ¬i:

1	$\phi \to \psi$	premise
2	$\neg\psi$	premise
3	ϕ	assumption
4	ψ	→e 1, 3
5	$\bot$	¬e 4, 2
6	$\neg\phi$	¬i 3−5

We could now go back through the proofs in this chapter and replace applications of MT by this combination of →e, ¬e and ¬i. However, it is convenient to think of MT as a shorthand (or a macro).

The same holds for the rule

$$\frac{\phi}{\neg\neg\phi}\ \neg\neg\mathrm{i}.$$

It can be derived from the rules ¬i and ¬e, as follows:

1	ϕ	premise
2	$\neg\phi$	assumption
3	$\bot$	¬e 1, 2
4	$\neg\neg\phi$	¬i 2−3

There are (unboundedly) many such derived rules which we could write down. However, there is no point in making our calculus fat and unwieldy; and some purists would say that we should stick to a minimum set of rules, all of which are independent of each other. We don't take such a purist view. Indeed, the two derived rules we now introduce are extremely useful. You will find that they crop up frequently when doing exercises in natural deduction, so it is worth giving them names as derived rules. In the case of the second one, its derivation from the primitive proof rules is not very obvious.

The first one has the Latin name *reductio ad absurdum*. It means 'reduction to absurdity' and we will simply call it *proof by contradiction* (PBC for short). The rule says: if from $\neg\phi$ we obtain a contradiction, then we are entitled to deduce ϕ:

$$\frac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\ \mathrm{PBC}.$$

This rule looks rather similar to ¬i, except that the negation is in a different place. This is the clue to how to derive PBC from our basic proof rules. Suppose we have a proof of $\bot$ from $\neg\phi$. By →i, we can transform this into a proof of $\neg\phi \to \bot$ and proceed as follows:

1	$\neg\phi \to \bot$	given
2	$\neg\phi$	assumption
3	$\bot$	→e 1, 2
4	$\neg\neg\phi$	¬i 2–3
5	ϕ	¬¬e 4

This shows that PBC can be derived from →i, ¬i, →e and ¬¬e.

The final derived rule we consider in this section is arguably the most useful to use in proofs, because its derivation is rather long and complicated, so its usage often saves time and effort. It also has a Latin name, *tertium non datur*; the English name is the law of the excluded middle, or LEM for short. It simply says that $\phi \lor \neg\phi$ is true: whatever ϕ is, it must be either true or false; in the latter case, $\neg\phi$ is true. There is no third possibility (hence *excluded middle*): the sequent $\vdash \phi \lor \neg\phi$ is valid. Its validity is implicit, for example, whenever you write an if-statement in a programming language: '`if` B $\{C_1\}$ `else` $\{C_2\}$' relies on the fact that $B \lor \neg B$ is always true (and that B and $\neg B$ can never be true at the same time). Here is a proof in natural deduction that derives the law of the excluded middle from basic proof rules:

1	$\neg(\phi \lor \neg\phi)$	assumption
2	ϕ	assumption
3	$\phi \lor \neg\phi$	$\lor\mathrm{i}_1$ 2
4	$\bot$	¬e 3, 1
5	$\neg\phi$	¬i 2–4
6	$\phi \lor \neg\phi$	$\lor\mathrm{i}_2$ 5
7	$\bot$	¬e 6, 1
8	$\neg\neg(\phi \lor \neg\phi)$	¬i 1–7
9	$\phi \lor \neg\phi$	¬¬e 8

Example 1.24 Using LEM, we show that $p \to q \vdash \neg p \lor q$ is valid:

1	$p \to q$	premise
2	$\neg p \lor p$	LEM
3	$\neg p$	assumption
4	$\neg p \lor q$	$\lor$i$_1$ 3
5	p	assumption
6	q	$\to$e $1, 5$
7	$\neg p \lor q$	$\lor$i$_2$ 6
8	$\neg p \lor q$	$\lor$e $2, 3{-}4, 5{-}7$

It can be difficult to decide which instance of LEM would benefit the progress of a proof. Can you re-do the example above with $q \lor \neg q$ as LEM?

1.2.3 Natural deduction in summary

The proof rules for natural deduction are summarised in Figure 1.2. The explanation of the rules we have given so far in this chapter is *declarative*; we have presented each rule and justified it in terms of our intuition about the logical connectives. However, when you try to use the rules yourself, you'll find yourself looking for a more *procedural* interpretation; what does a rule do and how do you use it? For example,

- $\land$i says: to prove $\phi \land \psi$, you must first prove ϕ and ψ separately and then use the rule $\land$i.
- $\land$e$_1$ says: to prove ϕ, try proving $\phi \land \psi$ and then use the rule $\land$e$_1$. Actually, this doesn't sound like very good advice because probably proving $\phi \land \psi$ will be harder than proving ϕ alone. However, you might find that you *already have* $\phi \land \psi$ lying around, so that's when this rule is useful. Compare this with the example sequent in Example 1.15.
- $\lor$i$_1$ says: to prove $\phi \lor \psi$, try proving ϕ. Again, in general it is harder to prove ϕ than it is to prove $\phi \lor \psi$, so this will usually be useful only if you've already managed to prove ϕ. For example, if you want to prove $q \vdash p \lor q$, you certainly won't be able simply to use the rule $\lor$i$_1$, but $\lor$i$_2$ will work.
- $\lor$e has an excellent procedural interpretation. It says: if you have $\phi \lor \psi$, and you want to prove some χ, then try to prove χ from ϕ and from ψ in turn. (In those subproofs, of course you can use the other prevailing premises as well.)
- Similarly, $\to$i says, if you want to prove $\phi \to \psi$, try proving ψ from ϕ (and the other prevailing premises).
- $\neg$i says: to prove $\neg\phi$, prove $\bot$ from ϕ (and the other prevailing premises).

The basic rules of natural deduction:

	introduction	*elimination*
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\,\wedge\mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\,\wedge\mathrm{e}_1 \qquad \dfrac{\phi \wedge \psi}{\psi}\,\wedge\mathrm{e}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\,\vee\mathrm{i}_1 \qquad \dfrac{\psi}{\phi \vee \psi}\,\vee\mathrm{i}_2$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\,\vee\mathrm{e}$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \rightarrow \psi}\,\rightarrow\mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\,\rightarrow\mathrm{e}$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}\,\neg\mathrm{i}$	$\dfrac{\phi \quad \neg\phi}{\bot}\,\neg\mathrm{e}$
$\bot$	(no introduction rule for $\bot$)	$\dfrac{\bot}{\phi}\,\bot\mathrm{e}$
$\neg\neg$		$\dfrac{\neg\neg\phi}{\phi}\,\neg\neg\mathrm{e}$

Some useful derived rules:

$$\dfrac{\phi \rightarrow \psi \quad \neg\psi}{\neg\phi}\,\mathrm{MT} \qquad\qquad \dfrac{\phi}{\neg\neg\phi}\,\neg\neg\mathrm{i}$$

$$\dfrac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\,\mathrm{PBC} \qquad\qquad \dfrac{}{\phi \vee \neg\phi}\,\mathrm{LEM}$$

Figure 1.2. Natural deduction rules for propositional logic.

At any stage of a proof, it is permitted to introduce any formula as assumption, by choosing a proof rule that opens a box. As we saw, natural deduction employs boxes to control the scope of assumptions. When an assumption is introduced, a box is opened. Discharging assumptions is achieved by closing a box according to the pattern of its particular proof rule. It's useful to make assumptions by opening boxes. *But don't forget you have to close them in the manner prescribed by their proof rule.*

OK, but how do we actually go about constructing a proof? Given a sequent, you write its premises at the top of your page and its conclusion at the bottom. Now, you're trying to fill in the gap, which involves working simultaneously on the premises (to bring them towards the conclusion) and on the conclusion (to massage it towards the premises).

Look first at the conclusion. If it is of the form $\phi \to \psi$, then apply[6] the rule →i. This means drawing a box with ϕ at the top and ψ at the bottom. So your proof, which started out like this:

$$\begin{array}{c} \vdots \\ \text{premises} \\ \vdots \\ \phi \to \psi \end{array}$$

now looks like this:

$$\begin{array}{ll} & \vdots \\ & \text{premises} \\ & \vdots \\ & \boxed{\begin{array}{lr} \phi & \qquad\qquad \text{assumption} \\ & \\ & \\ \psi & \end{array}} \\ \phi \to \psi & \qquad \to\text{i} \end{array}$$

You still have to find a way of filling in the gap between the ϕ and the ψ. But you now have an extra formula to work with and you have simplified the conclusion you are trying to reach.

[6] Except in situations such as $p \to (q \to \neg r), p \vdash q \to \neg r$ where →e produces a simpler proof.

The proof rule ¬i is very similar to →i and has the same beneficial effect on your proof attempt. It gives you an extra premise to work with and simplifies your conclusion.

At any stage of a proof, several rules are likely to be applicable. Before applying any of them, list the applicable ones and think about which one is likely to improve the situation for your proof. You'll find that →i and ¬i most often improve it, so always use them whenever you can. There is no easy recipe for when to use the other rules; often you have to make judicious choices.

1.2.4 Provable equivalence

Definition 1.25 Let ϕ and ψ be formulas of propositional logic. We say that ϕ and ψ are *provably equivalent* iff (we write 'iff' for 'if, and only if' in the sequel) the sequents $\phi \vdash \psi$ and $\psi \vdash \phi$ are valid; that is, there is a proof of ψ from ϕ and another one going the other way around. As seen earlier, we denote that ϕ and ψ are provably equivalent by $\phi \dashv\vdash \psi$.

Note that, by Remark 1.12, we could just as well have defined $\phi \dashv\vdash \psi$ to mean that the sequent $\vdash (\phi \to \psi) \land (\psi \to \phi)$ is valid; it defines the same concept. Examples of provably equivalent formulas are

$$\begin{array}{ll}
\neg(p \land q) \dashv\vdash \neg q \lor \neg p & \neg(p \lor q) \dashv\vdash \neg q \land \neg p \\
p \to q \dashv\vdash \neg q \to \neg p & p \to q \dashv\vdash \neg p \lor q \\
p \land q \to p \dashv\vdash r \lor \neg r & p \land q \to r \dashv\vdash p \to (q \to r).
\end{array}$$

The reader should prove all of these six equivalences in natural deduction.

1.2.5 An aside: proof by contradiction

Sometimes we can't prove something *directly* in the sense of taking apart given assumptions and reasoning with their constituents in a constructive way. Indeed, the proof system of natural deduction, summarised in Figure 1.2, specifically allows for *indirect* proofs that lack a constructive quality: for example, the rule

$$\frac{\boxed{\begin{array}{c}\neg\phi \\ \vdots \\ \bot\end{array}}}{\phi}\ \mathrm{PBC}$$

allows us to prove ϕ by showing that $\neg\phi$ leads to a contradiction. Although 'classical logicians' argue that this is valid, logicians of another kind, called 'intuitionistic logicians,' argue that to prove ϕ you should do it directly, rather than by arguing merely that $\neg\phi$ is impossible. The two other rules on which classical and intuitionistic logicians disagree are

$$\frac{}{\phi \vee \neg\phi}\ \text{LEM} \qquad \frac{\neg\neg\phi}{\phi}\ \neg\neg\text{e}.$$

Intuitionistic logicians argue that, to show $\phi \vee \neg\phi$, you have to show ϕ, or $\neg\phi$. If neither of these can be shown, then the putative truth of the disjunction has no justification. Intuitionists reject $\neg\neg$e since we have already used this rule to prove LEM and PBC from rules which the intuitionists do accept. In the exercises, you are asked to show why the intuitionists also reject PBC.

Let us look at a proof that shows up this difference, involving real numbers. Real numbers are floating point numbers like 23.54721, only some of them might actually be infinitely long such as 23.138592748500123950734..., with no periodic behaviour after the decimal point.

Given a positive real number a and a *natural* (whole) number b, we can calculate a^b: it is just a times itself, b times, so $2^2 = 2 \cdot 2 = 4$, $2^3 = 2 \cdot 2 \cdot 2 = 8$ and so on. When b is a *real* number, we can also define a^b, as follows. We say that $a^0 \stackrel{\text{def}}{=} 1$ and, for a non-zero rational number k/n, where $n \neq 0$, we let $a^{k/n} \stackrel{\text{def}}{=} \sqrt[n]{a^k}$ where $\sqrt[n]{x}$ is the real number y such that $y^n = x$. From real analysis one knows that any real number b can be approximated by a sequence of rational numbers k_0/n_0, k_1/n_1, ... Then we define a^b to be the real number approximated by the sequence a^{k_0/n_0}, a^{k_1/n_1}, ... (In calculus, one can show that this 'limit' a^b is unique and independent of the choice of approximating sequence.) Also, one calls a real number *irrational* if it can't be written in the form k/n for some integers k and $n \neq 0$. In the exercises you will be asked to find a semi-formal proof showing that $\sqrt{2}$ is irrational.

We now present a proof of a fact about real numbers in the informal style used by mathematicians (this proof can be formalised as a natural deduction proof in the logic presented in Chapter 2). The fact we prove is:

Theorem 1.26 *There exist irrational numbers a and b such that a^b is rational.*

PROOF: We choose b to be $\sqrt{2}$ and proceed by a case analysis. Either b^b is irrational, or it is not. (Thus, our proof uses $\vee$e on an instance of LEM.)

(i) Assume that b^b is rational. Then this proof is easy since we can choose irrational numbers a and b to be $\sqrt{2}$ and see that a^b is just b^b which was assumed to be rational.

(ii) Assume that b^b is *ir*rational. Then we change our strategy slightly and choose a to be $\sqrt{2}^{\sqrt{2}}$. Clearly, a is irrational by the assumption of case (ii). But we know that b is irrational (this was known by the ancient Greeks; see the proof outline in the exercises). So a and b are both irrational numbers and

$$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = (\sqrt{2})^2 = 2$$

is rational, where we used the law $(x^y)^z = x^{(y\cdot z)}$.

Since the two cases above are exhaustive (*either* b^b is irrational, *or* it isn't) we have proven the theorem. □

This proof is perfectly legitimate and mathematicians use arguments like that all the time. The exhaustive nature of the case analysis above rests on the use of the rule LEM, which we use to prove that either b is rational or it is not. Yet, there is something puzzling about it. Surely, we have secured the fact that there are irrational numbers a and b such that a^b is rational, but are we in a position to specify an actual pair of such numbers satisfying this theorem? More precisely, which of the pairs (a, b) above fulfils the assertion of the theorem, the pair $(\sqrt{2}, \sqrt{2})$, or the pair $(\sqrt{2}^{\sqrt{2}}, \sqrt{2})$? Our proof tells us nothing about *which* of them is the right choice; it just says that at least one of them works.

Thus, the intuitionists favour a calculus containing the introduction and elimination rules shown in Figure 1.2 and excluding the rule ¬¬e and the derived rules. Intuitionistic logic turns out to have some specialised applications in computer science, such as modelling type-inference systems used in compilers or the staged execution of program code; but in this text we stick to the full so-called classical logic which includes all the rules.

1.3 Propositional logic as a formal language

In the previous section we learned about propositional atoms and how they can be used to build more complex logical formulas. We were deliberately informal about that, for our main focus was on trying to understand the precise mechanics of the natural deduction rules. However, it should have been clear that the rules we stated are valid for *any* formulas we can form, as long as they match the pattern required by the respective rule. For example, the application of the proof rule →e in

1	$p \rightarrow q$	premise
2	p	premise
3	q	$\rightarrow$e $1, 2$

is equally valid if we substitute p with $p \vee \neg r$ and q with $r \rightarrow p$:

1	$p \vee \neg r \rightarrow (r \rightarrow p)$	premise
2	$p \vee \neg r$	premise
3	$r \rightarrow p$	$\rightarrow$e $1, 2$

This is why we expressed such rules as schemes with Greek symbols standing for generic formulas. Yet, it is time that we make precise the notion of 'any formula we may form.' Because this text concerns various logics, we will introduce in (1.3) an easy formalism for specifying well-formed formulas. In general, we need an *unbounded* supply of propositional atoms $p, q, r, \ldots$, or $p_1, p_2, p_3, \ldots$ You should not be too worried about the need for infinitely many such symbols. Although we may only need *finitely many* of these propositions to describe a property of a computer program successfully, we cannot specify how many such atomic propositions we will need in any concrete situation, so having infinitely many symbols at our disposal is a cheap way out. This can be compared with the potentially infinite nature of English: the number of grammatically correct English sentences is infinite, but finitely many such sentences will do in whatever situation you might be in (writing a book, attending a lecture, listening to the radio, having a dinner date, ...).

Formulas in our propositional logic should certainly be strings over the alphabet $\{p, q, r, \ldots\} \cup \{p_1, p_2, p_3, \ldots\} \cup \{\neg, \wedge, \vee, \rightarrow, (,)\}$. This is a trivial observation and as such is not good enough for what we are trying to capture. For example, the string $(\neg)() \vee pq \rightarrow$ is a word over that alphabet, yet, it does not seem to make a lot of sense as far as propositional logic is concerned. So what we have to define are those strings which we want to call formulas. We call such formulas *well-formed.*

Definition 1.27 The well-formed formulas of propositional logic are those which we obtain by using the construction rules below, and only those, finitely many times:

atom: Every propositional atom $p, q, r, \dots$ and $p_1, p_2, p_3, \dots$ is a well-formed formula.

$\neg$: If ϕ is a well-formed formula, then so is $(\neg\phi)$.

$\wedge$: If ϕ and ψ are well-formed formulas, then so is $(\phi \wedge \psi)$.

$\vee$: If ϕ and ψ are well-formed formulas, then so is $(\phi \vee \psi)$.

$\rightarrow$: If ϕ and ψ are well-formed formulas, then so is $(\phi \rightarrow \psi)$.

It is most crucial to realize that this definition is the one a computer would expect and that we did not make use of the binding priorities agreed upon in the previous section.

Convention. In this section we act as if we are a rigorous computer and we call formulas well-formed iff they can be deduced to be so using the definition above.

Further, note that the condition 'and only those' in the definition above rules out the possibility of any other means of establishing that formulas are well-formed. Inductive definitions, like the one of well-formed propositional logic formulas above, are so frequent that they are often given by a defining grammar in Backus Naur form (BNF). In that form, the above definition reads more compactly as

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \tag{1.3}$$

where p stands for any atomic proposition and each occurrence of ϕ to the right of ::= stands for any already constructed formula.

So how can we show that a string is a well-formed formula? For example, how do we answer this for ϕ being

$$(((\neg p) \wedge q) \rightarrow (p \wedge (q \vee (\neg r)))) \ ? \tag{1.4}$$

Such reasoning is greatly facilitated by the fact that the grammar in (1.3) satisfies the *inversion principle*, which means that we can invert the process of building formulas: although the grammar rules allow for five different ways of constructing more complex formulas – the five clauses in (1.3) – there is always a unique clause which was used last. For the formula above, this last operation was an application of the fifth clause, for ϕ is an implication with the assumption $((\neg p) \wedge q)$ and conclusion $(p \wedge (q \vee (\neg r)))$. By applying the inversion principle to the assumption, we see that it is a conjunction of $(\neg p)$ and q. The former has been constructed using the second clause and is well-formed since p is well-formed by the first clause in (1.3). The latter is well-formed for the same reason. Similarly, we can apply the inversion

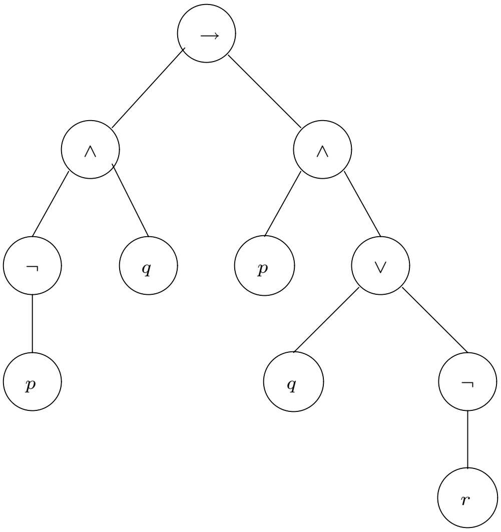


Figure 1.3. A parse tree representing a well-formed formula.

principle to the conclusion $(p \wedge (q \vee (\neg r)))$, inferring that it is indeed well-formed. In summary, the formula in (1.4) is well-formed.

For us humans, dealing with brackets is a tedious task. The reason we need them is that formulas really have a tree-like structure, although we prefer to represent them in a linear way. In Figure 1.3 you can see the parse tree[7] of the well-formed formula ϕ in (1.4). Note how brackets become unnecessary in this parse tree since the paths and the branching structure of this tree remove any possible ambiguity in interpreting ϕ. In representing ϕ as a linear string, the branching structure of the tree is retained by the insertion of brackets as done in the definition of well-formed formulas.

So how would you go about showing that a string of symbols ψ is *not* well-formed? At first sight, this is a bit trickier since we somehow have to make sure that ψ could not have been obtained by *any* sequence of construction rules. Let us look at the formula $(\neg)() \vee pq \rightarrow$ from above. We can decide this matter by being very observant. The string $(\neg)() \vee pq \rightarrow$ contains $\neg)$ and $\neg$ cannot be the rightmost symbol of a well-formed formula (check all the rules to verify this claim!); but the only time we can put a ')' to the right of something is if that something is a well-formed formula (again, check all the rules to see that this is so). Thus, $(\neg)() \vee pq \rightarrow$ is *not* well-formed.

Probably the easiest way to verify whether some formula ϕ is well-formed is by trying to draw its parse tree. In this way, you can verify that the

[7] We will use this name without explaining it any further and are confident that you will understand its meaning through the examples.

formula in (1.4) is well-formed. In Figure 1.3 we see that its parse tree has $\rightarrow$ as its root, expressing that the formula is, at its top level, an implication. Using the grammar clause for implication, it suffices to show that the left and right subtrees of this root node are well-formed. That is, we proceed in a top-down fashion and, in this case, successfully. Note that the parse trees of well-formed formulas have either an atom as root (and then this is all there is in the tree), or the root contains $\neg$, $\vee$, $\wedge$ or $\rightarrow$. In the case of $\neg$ there is only *one* subtree coming out of the root. In the cases $\wedge$, $\vee$ or $\rightarrow$ we must have *two* subtrees, each of which must behave as just described; this is another example of an *inductive* definition.

Thinking in terms of trees will help you understand standard notions in logic, for example, the concept of a *subformula*. Given the well-formed formula ϕ above, its subformulas are just the ones that correspond to the subtrees of its parse tree in Figure 1.3. So we can list all its leaves p, q (occurring twice), and r, then $(\neg p)$ and $((\neg p) \wedge q)$ on the left subtree of $\rightarrow$ and $(\neg r)$, $(q \vee (\neg r))$ and $((p \wedge (q \vee (\neg p))))$ on the right subtree of $\rightarrow$. The whole tree is a subtree of itself as well. So we can list all nine subformulas of ϕ as

$$\begin{array}{l}
p \\
q \\
r \\
(\neg p) \\
((\neg p) \wedge q) \\
(\neg r) \\
(q \vee (\neg r)) \\
((p \wedge (q \vee (\neg r)))) \\
(((\neg p) \wedge q) \rightarrow (p \wedge (q \vee (\neg r)))).
\end{array}$$

Let us consider the tree in Figure 1.4. Why does it represent a well-formed formula? All its leaves are propositional atoms (p twice, q and r), all branching nodes are logical connectives ($\neg$ twice, $\wedge$, $\vee$ and $\rightarrow$) and the numbers of subtrees are correct in all those cases (one subtree for a $\neg$ node and two subtrees for all other non-leaf nodes). How do we obtain the linear representation of this formula? If we ignore brackets, then we are seeking nothing but the *in-order* representation of this tree as a list[8]. The resulting well-formed formula is $((\neg(p \vee (q \rightarrow (\neg p)))) \wedge r)$.

[8] The other common ways of flattening trees to lists are *preordering* and *postordering*. See any text on binary trees as data structures for further details.

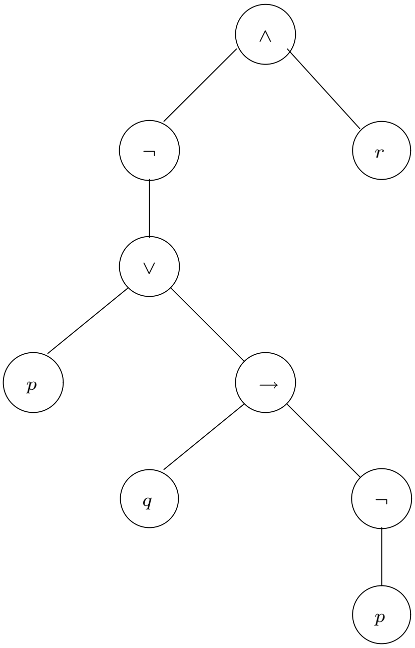


Figure 1.4. Given: a tree; wanted: its linear representation as a logical formula.

The tree in Figure 1.21 on page 82, however, does *not* represent a well-formed formula for two reasons. First, the leaf $\wedge$ (and a similar argument applies to the leaf $\neg$), the left subtree of the node $\rightarrow$, is not a propositional atom. This could be fixed by saying that we decided to leave the left and right subtree of that node unspecified and that we are willing to provide those now. However, the second reason is fatal. The p node is not a leaf since it has a subtree, the node $\neg$. This cannot make sense if we think of the entire tree as some logical formula. So this tree does not represent a well-formed logical formula.

1.4 Semantics of propositional logic

1.4.1 The meaning of logical connectives

In the second section of this chapter, we developed a calculus of reasoning which could verify that sequents of the form $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ are valid, which means: from the premises ϕ_1, ϕ_2, …, ϕ_n, we may conclude ψ.

In this section we give another account of this relationship between the premises ϕ_1, ϕ_2, …, ϕ_n and the conclusion ψ. To contrast with the sequent above, we define a new relationship, written

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

This account is based on looking at the 'truth values' of the atomic formulas in the premises and the conclusion; and at how the logical connectives manipulate these truth values. What is the truth value of a declarative sentence, like sentence (3) 'Every even natural number > 2 is the sum of two prime numbers'? Well, declarative sentences express a fact about the real world, the physical world we live in, or more abstract ones such as computer models, or our thoughts and feelings. Such factual statements either match reality (they are *true*), or they don't (they are *false*).

If we combine declarative sentences p and q with a logical connective, say $\land$, then the truth value of $p \land q$ is determined by three things: the truth value of p, the truth value of q and the meaning of $\land$. The meaning of $\land$ is captured by the observation that $p \land q$ is true iff p *and* q are both true; otherwise $p \land q$ is false. Thus, as far as $\land$ is concerned, it needs only to know whether p and q are true, it does *not* need to know what p and q are actually saying about the world out there. This is also the case for all the other logical connectives and is the reason why we can compute the truth value of a formula just by knowing the truth values of the atomic propositions occurring in it.

Definition 1.28 1. The set of truth values contains two elements `T` and `F`, where `T` represents 'true' and `F` represents 'false'.

2. A *valuation* or *model* of a formula ϕ is an assignment of each propositional atom in ϕ to a truth value.

Example 1.29 The map which assigns `T` to q and `F` to p is a valuation for $p \lor \neg q$. Please list the remaining three valuations for this formula.

We can think of the meaning of $\land$ as a function of two arguments; each argument is a truth value and the result is again such a truth value. We specify this function in a table, called the *truth table for conjunction*, which you can see in Figure 1.5. In the first column, labelled ϕ, we list all possible

ϕ	ψ	$\phi \land \psi$
T	T	T
T	F	F
F	T	F
F	F	F

Figure 1.5. The truth table for conjunction, the logical connective $\land$.

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

ϕ	ψ	$\phi \to \psi$
T	T	T
T	F	F
F	T	T
F	F	T

ϕ	$\neg\phi$
T	F
F	T

$\top$
T

$\bot$
F

Figure 1.6. The truth tables for all the logical connectives discussed so far.

truth values of ϕ. Actually we list them *twice* since we also have to deal with another formula ψ, so the possible number of combinations of truth values for ϕ and ψ equals $2 \cdot 2 = 4$. Notice that the four pairs of ϕ and ψ values in the first two columns really exhaust all those possibilities (TT, TF, FT and FF). In the third column, we list the result of $\phi \wedge \psi$ according to the truth values of ϕ and ψ. So in the first line, where ϕ and ψ have value T, the result is T again. In all other lines, the result is F since at least one of the propositions ϕ or ψ has value F.

In Figure 1.6 you find the truth tables for all logical connectives of propositional logic. Note that $\neg$ turns T into F and vice versa. Disjunction is the mirror image of conjunction if we swap T and F, namely, a disjunction returns F iff both arguments are equal to F, otherwise (= at least one of the arguments equals T) it returns T. The behaviour of implication is not quite as intuitive. Think of the meaning of $\to$ as checking whether *truth is being preserved*. Clearly, this is not the case when we have $\mathtt{T} \to \mathtt{F}$, since we infer something that is false from something that is true. So the second entry in the column $\phi \to \psi$ equals F. On the other hand, $\mathtt{T} \to \mathtt{T}$ obviously preserves truth, but so do the cases $\mathtt{F} \to \mathtt{T}$ and $\mathtt{F} \to \mathtt{F}$, because there is no truth to be preserved in the first place as the assumption of the implication is false.

If you feel slightly uncomfortable with the semantics (= the meaning) of $\to$, then it might be good to think of $\phi \to \psi$ as an abbreviation of the formula $\neg\phi \vee \psi$ *as far as meaning is concerned*; these two formulas are very different syntactically and natural deduction treats them differently as well. But using the truth tables for $\neg$ and $\vee$ you can check that $\phi \to \psi$ evaluates to T iff $\neg\phi \vee \psi$ does so. This means that $\phi \rightarrow \psi$ and $\neg\phi \vee \psi$ are *semantically equivalent*; more on that in Section 1.5.

Given a formula ϕ which contains the propositional atoms $p_1, p_2, \ldots, p_n$, we can construct a truth table for ϕ, at least in principle. The caveat is that this truth table has 2^n many lines, each line listing a possible combination of truth values for $p_1, p_2, \ldots, p_n$; and for large n this task is impossible to complete. Our aim is thus to compute the value of ϕ for each of these 2^n cases for moderately small values of n. Let us consider the example ϕ in Figure 1.3. It involves three propositional atoms ($n = 3$) so we have $2^3 = 8$ cases to consider.

We illustrate how things go for one particular case, namely for the valuation in which q evaluates to F; and p and r evaluate to T. What does $\neg p \wedge q \rightarrow p \wedge (q \vee \neg r)$ evaluate to? Well, the beauty of our semantics is that it is *compositional.* If we know the meaning of the subformulas $\neg p \wedge q$ and $p \wedge (q \vee \neg r)$, then we just have to look up the appropriate line of the $\rightarrow$ truth table to find the value of ϕ, for ϕ is an implication of these two subformulas. Therefore, we can do the calculation by traversing the parse tree of ϕ in a bottom-up fashion. We know what its leaves evaluate to since we stated what the atoms p, q and r evaluated to. Because the meaning of p is T, we see that $\neg p$ computes to F. Now q is assumed to represent F and the conjunction of F and F is F. Thus, the left subtree of the node $\rightarrow$ evaluates to F. As for the right subtree of $\rightarrow$, r stands for T so $\neg r$ computes to F and q means F, so the disjunction of F and F is still F. We have to take that result, F, and compute its conjunction with the meaning of p which is T. Since the conjunction of T and F is F, we get F as the meaning of the right subtree of $\rightarrow$. Finally, to evaluate the meaning of ϕ, we compute F $\rightarrow$ F which is T. Figure 1.7 shows how the truth values propagate upwards to reach the root whose associated truth value is the truth value of ϕ given the meanings of p, q and r above.

It should now be quite clear how to build a truth table for more complex formulas. Figure 1.8 contains a truth table for the formula $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$. To be more precise, the first two columns list all possible combinations of values for p and q. The next two columns compute the corresponding values for $\neg p$ and $\neg q$. Using these four columns, we may compute the column for $p \rightarrow \neg q$ and $q \vee \neg p$. To do so we think of the first and fourth columns as the data for the $\rightarrow$ truth table and compute the column of $p \rightarrow \neg q$ accordingly. For example, in the first line p is T and $\neg q$ is F so the entry for $p \rightarrow \neg q$ is $\mathtt{T} \rightarrow \mathtt{F} = \mathtt{F}$ by definition of the meaning of $\rightarrow$. In this fashion, we can fill out the rest of the fifth column. Column 6 works similarly, only we now need to look up the truth table for $\vee$ with columns 2 and 3 as input.

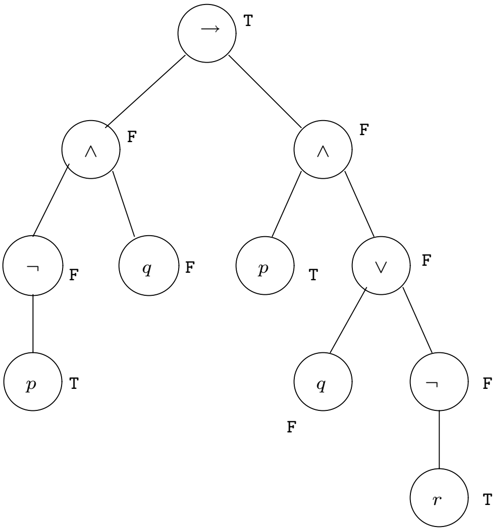


Figure 1.7. The evaluation of a logical formula under a given valuation.

p	q	$\neg p$	$\neg q$	$p \to \neg q$	$q \vee \neg p$	$(p \to \neg q) \to (q \vee \neg p)$
T	T	F	F	F	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	T
F	F	T	T	T	T	T

Figure 1.8. An example of a truth table for a more complex logical formula.

Finally, column 7 results from applying the truth table of → to columns 5 and 6.

1.4.2 Mathematical induction

Here is a little anecdote about the German mathematician Gauss who, as a pupil at age 8, did not pay attention in class (can you imagine?), with the result that his teacher made him sum up all natural numbers from 1 to 100. The story has it that Gauss came up with the correct answer 5050 within seconds, which infuriated his teacher. How did Gauss do it? Well, possibly he knew that

$$1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n+1)}{2} \tag{1.5}$$

for all natural numbers n.[9] Thus, taking $n = 100$, Gauss could easily calculate:

$$1 + 2 + 3 + 4 + \cdots + 100 = \frac{100 \cdot 101}{2} = 5050.$$

Mathematical induction allows us to prove equations, such as the one in (1.5), for arbitrary n. More generally, it allows us to show that *every* natural number satisfies a certain property. Suppose we have a property M which we think is true of all natural numbers. We write $M(5)$ to say that the property is true of 5, etc. Suppose that we know the following two things about the property M:

1. **Base case:** The natural number 1 has property M, i.e. we have a proof of $M(1)$.
2. **Inductive step:** If n is a natural number which *we assume* to have property $M(n)$, then *we can show* that $n + 1$ has property $M(n + 1)$; i.e. we have a proof of $M(n) \to M(n + 1)$.

Definition 1.30 The principle of mathematical induction says that, on the grounds of these two pieces of information above, every natural number n has property $M(n)$. The assumption of $M(n)$ in the inductive step is called the *induction hypothesis.*

Why does this principle make sense? Well, take *any* natural number k. If k equals 1, then k has property $M(1)$ using the base case and so we are done. Otherwise, we can use the inductive step, applied to $n = 1$, to infer that $2 = 1 + 1$ has property $M(2)$. We can do that using $\to$e, for we know that 1 has the property in question. Now we use that same inductive step on $n = 2$ to infer that 3 has property $M(3)$ and we repeat this until we reach $n = k$ (see Figure 1.9). Therefore, we should have no objections about using the principle of mathematical induction for natural numbers.

Returning to Gauss' example we claim that the sum $1 + 2 + 3 + 4 + \cdots + n$ equals $n \cdot (n + 1)/2$ for all natural numbers n.

Theorem 1.31 *The sum* $1 + 2 + 3 + 4 + \cdots + n$ *equals* $n \cdot (n + 1)/2$ *for all natural numbers* n*.*

[9] There is another way of finding the sum $1 + 2 + \cdots + 100$, which works like this: write the sum backwards, as $100 + 99 + \cdots + 1$. Now add the forwards and backwards versions, obtaining $101 + 101 + \cdots + 101$ (100 times), which is 10100. Since we added the sum to itself, we now divide by two to get the answer 5050. Gauss probably used this method; but the method of mathematical induction that we explore in this section is much more powerful and can be applied in a wide variety of situations.

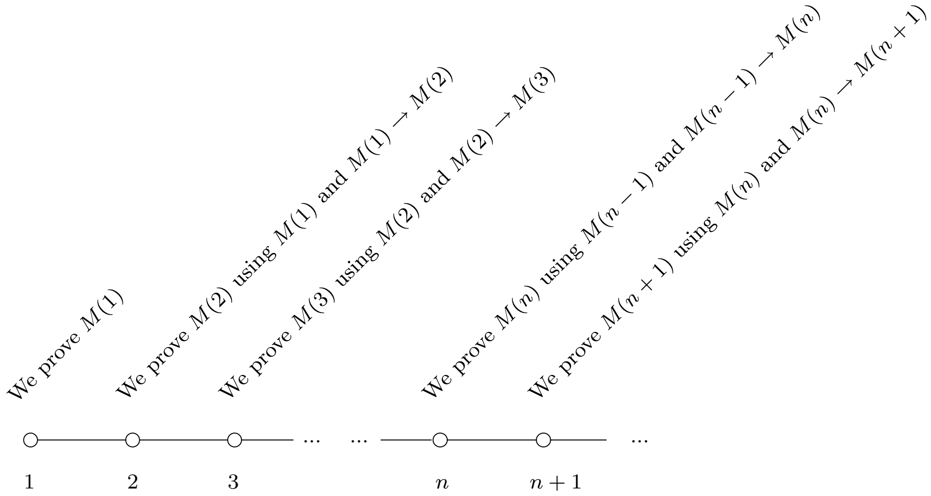

We prove $M(1)$
We prove $M(2)$ using $M(1)$ and $M(1) \to M(2)$
We prove $M(3)$ using $M(2)$ and $M(2) \to M(3)$
We prove $M(n)$ using $M(n-1)$ and $M(n-1) \to M(n)$
We prove $M(n+1)$ using $M(n)$ and $M(n) \to M(n+1)$
1
2
3
n
$n+1$

Figure 1.9. How the principle of mathematical induction works. By proving just two facts, $M(1)$ and $M(n) \to M(n+1)$ for a formal (and unconstrained) parameter n, we are able to deduce $M(k)$ for each natural number k.

PROOF: We use mathematical induction. In order to reveal the fine structure of our proof we write LHS_n for the expression $1+2+3+4+\cdots+n$ and RHS_n for $n \cdot (n+1)/2$. Thus, we need to show $\mathrm{LHS}_n = \mathrm{RHS}_n$ for all $n \geq 1$.

Base case: If n equals 1, then LHS_1 is just 1 (there is only one summand), which happens to equal $\mathrm{RHS}_1 = 1 \cdot (1+1)/2$.

Inductive step: Let us assume that $\mathrm{LHS}_n = \mathrm{RHS}_n$. Recall that this assumption is called the induction hypothesis; it is the driving force of our argument. We need to show $\mathrm{LHS}_{n+1} = \mathrm{RHS}_{n+1}$, i.e. that the longer sum $1+2+3+4+\cdots+(n+1)$ equals $(n+1)\cdot((n+1)+1)/2$. The key observation is that the sum $1+2+3+4+\cdots+(n+1)$ is nothing but the sum $(1+2+3+4+\cdots+n)+(n+1)$ of two summands, where the first one is the sum of our induction hypothesis. The latter says that $1+2+3+4+\cdots+n$ equals $n\cdot(n+1)/2$, and we are certainly entitled to substitute equals for equals in our reasoning. Thus, we compute

$$\begin{aligned}
&\mathrm{LHS}_{n+1}\\
&\qquad = 1+2+3+4+\cdots+(n+1)\\
&\qquad = \mathrm{LHS}_n + (n+1) \quad \text{regrouping the sum}
\end{aligned}$$

$$
\begin{aligned}
&= \mathrm{RHS}_n + (n+1) \quad \text{by our induction hypothesis}\\
&= \tfrac{n\cdot(n+1)}{2} + (n+1)\\
&= \tfrac{n\cdot(n+1)}{2} + \tfrac{2\cdot(n+1)}{2} \quad \text{arithmetic}\\
&= \tfrac{(n+2)\cdot(n+1)}{2} \quad \text{arithmetic}\\
&= \tfrac{((n+1)+1)\cdot(n+1)}{2} \quad \text{arithmetic}\\
&= \mathrm{RHS}_{n+1}.
\end{aligned}
$$

Since we successfully showed the base case and the inductive step, we can use mathematical induction to infer that all natural numbers n have the property stated in the theorem above. □

Actually, there are numerous variations of this principle. For example, we can think of a version in which the base case is $n = 0$, which would then cover all natural numbers including 0. Some statements hold only for all natural numbers, say, greater than 3. So you would have to deal with a base case 4, but keep the version of the inductive step (see the exercises for such an example). The use of mathematical induction typically suceeds on properties $M(n)$ that involve inductive definitions (e.g. the definition of k^l with $l \geq 0$). Sentence (3) on page 2 suggests there may be true properties $M(n)$ for which mathematical induction won't work.

Course-of-values induction. There is a variant of mathematical induction in which the induction hypothesis for proving $M(n+1)$ is not just $M(n)$, but the conjunction $M(1) \wedge M(2) \wedge \cdots \wedge M(n)$. In that variant, called *course-of-values* induction, there doesn't have to be an explicit base case at all – everything can be done in the inductive step.

How can this work without a base case? The answer is that the base case is implicitly included in the inductive step. Consider the case $n = 3$: the inductive-step instance is $M(1) \wedge M(2) \wedge M(3) \to M(4)$. Now consider $n = 1$: the inductive-step instance is $M(1) \to M(2)$. What about the case when n equals 0? In this case, there are zero formulas on the left of the $\to$, so we have to prove $M(1)$ from nothing at all. The inductive-step instance is simply the obligation to show $M(1)$. You might find it useful to modify Figure 1.9 for course-of-values induction.

Having said that the base case is implicit in course-of-values induction, it frequently turns out that it still demands special attention when you get inside trying to prove the inductive case. We will see precisely this in the two applications of course-of-values induction in the following pages.

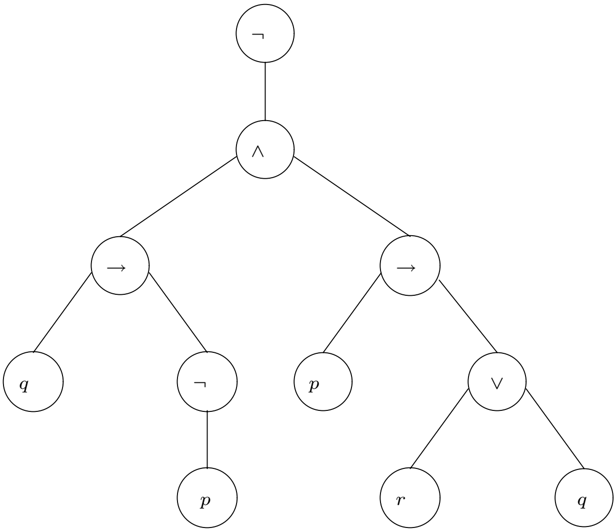


Figure 1.10. A parse tree with height 5.

In computer science, we often deal with finite structures of some kind, data structures, programs, files etc. Often we need to show that *every* instance of such a structure has a certain property. For example, the well-formed formulas of Definition 1.27 have the property that the number of '(' brackets in a particular formula equals its number of ')' brackets. We can use mathematical induction on the domain of natural numbers to prove this. In order to succeed, we somehow need to connect well-formed formulas to natural numbers.

Definition 1.32 Given a well-formed formula ϕ, we define its *height* to be 1 plus the length of the longest path of its parse tree.

For example, consider the well-formed formulas in Figures 1.3, 1.4 and 1.10. Their heights are 5, 6 and 5, respectively. In Figure 1.3, the longest path goes from $\rightarrow$ to $\wedge$ to $\vee$ to $\neg$ to r, a path of length 4, so the height is $4 + 1 = 5$. Note that the height of atoms is $1 + 0 = 1$. Since every well-formed formula has finite height, we can show statements about all well-formed formulas by mathematical induction on their height. This trick is most often called *structural induction*, an important reasoning technique in computer science. Using the notion of the height of a parse tree, we realise that structural induction is just a special case of course-of-values induction.

Theorem 1.33 *For every well-formed propositional logic formula, the number of left brackets is equal to the number of right brackets.*

PROOF: We proceed by course-of-values induction on the height of well-formed formulas ϕ. Let $M(n)$ mean 'All formulas of height n have the same number of left and right brackets.' We assume $M(k)$ for each $k < n$ and try to prove $M(n)$. Take a formula ϕ of height n.

- **Base case:** Then $n = 1$. This means that ϕ is just a propositional atom. So there are no left or right brackets, 0 equals 0.
- **Course-of-values inductive step:** Then $n > 1$ and so the root of the parse tree of ϕ must be $\neg$, $\rightarrow$, $\vee$ or $\wedge$, for ϕ is well-formed. We assume that it is $\rightarrow$, the other three cases are argued in a similar way. Then ϕ equals $(\phi_1 \rightarrow \phi_2)$ for some well-formed formulas ϕ_1 and ϕ_2 (of course, they are just the left, respectively right, linear representations of the root's two subtrees). It is clear that the heights of ϕ_1 and ϕ_2 are strictly smaller than n. Using the induction hypothesis, we therefore conclude that ϕ_1 has the same number of left and right brackets and that the same is true for ϕ_2. But in $(\phi_1 \rightarrow \phi_2)$ we added just two more brackets, one '(' and one ')'. Thus, the number of occurrences of '(' and ')' in ϕ is the same. □

The formula $(p \rightarrow (q \wedge \neg r))$ illustrates why we could not prove the above directly with mathematical induction on the height of formulas. While this formula has height 4, its two subtrees have heights 1 and 3, respectively. Thus, an induction hypothesis for height 3 would have worked for the right subtree but failed for the left subtree.

1.4.3 Soundness of propositional logic

The natural deduction rules make it possible for us to develop rigorous threads of argumentation, in the course of which we arrive at a conclusion ψ assuming certain other propositions $\phi_1, \phi_2, \dots, \phi_n$. In that case, we said that the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ is valid. Do we have any evidence that these rules are all *correct* in the sense that valid sequents all 'preserve truth' computed by our truth-table semantics?

Given a proof of $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$, is it conceivable that there is a valuation in which ψ above is false although all propositions $\phi_1, \phi_2, \dots, \phi_n$ are true? Fortunately, this is not the case and in this subsection we demonstrate why this is so. Let us suppose that some proof in our natural deduction calculus has established that the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ is valid. We need to show: for all valuations in which all propositions $\phi_1, \phi_2, \dots, \phi_n$ evaluate to $\mathtt{T}$, ψ evaluates to $\mathtt{T}$ as well.

Definition 1.34 If, for all valuations in which all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to $\mathtt{T}$, ψ evaluates to $\mathtt{T}$ as well, we say that

$$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$$

holds and call $\vDash$ the *semantic entailment* relation.

Let us look at some examples of this notion.

1. Does $p \wedge q \vDash p$ hold? Well, we have to inspect all assignments of truth values to p and q; there are four of these. Whenever such an assignment computes $\mathtt{T}$ for $p \wedge q$ we need to make sure that p is true as well. But $p \wedge q$ computes $\mathtt{T}$ only if p and q are true, so $p \wedge q \vDash p$ is indeed the case.
2. What about the relationship $p \vee q \vDash p$? There are three assignments for which $p \vee q$ computes $\mathtt{T}$, so p would have to be true for all of these. However, if we assign $\mathtt{T}$ to q and $\mathtt{F}$ to p, then $p \vee q$ computes $\mathtt{T}$, but p is false. Thus, $p \vee q \vDash p$ does not hold.
3. What if we modify the above to $\neg q, p \vee q \vDash p$? Notice that we have to be concerned only about valuations in which $\neg q$ *and* $p \vee q$ evaluate to $\mathtt{T}$. This forces q to be false, which in turn forces p to be true. Hence $\neg q, p \vee q \vDash p$ is the case.
4. Note that $p \vDash q \vee \neg q$ holds, despite the fact that no atomic proposition on the right of $\vDash$ occurs on the left of $\vDash$.

From the discussion above we realize that a soundness argument has to show: if $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds.

Theorem 1.35 (Soundness) *Let $\phi_1, \phi_2, \ldots, \phi_n$ and ψ be propositional logic formulas. If $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds.*

PROOF: Since $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid we know there is a proof of ψ from the premises $\phi_1, \phi_2, \ldots, \phi_n$. We now do a pretty slick thing, namely, we reason by *mathematical induction on the length of this proof!* The length of a proof is just the number of lines it involves. So let us be perfectly clear about what it is we mean to show. We intend to show the assertion $M(k)$:

> *'For all sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ $(n \geq 0)$ which have a proof of length k, it is the case that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds.'*

by course-of-values induction on the natural number k. This idea requires some work, though. The sequent $p \wedge q \to r \vdash p \to (q \to r)$ has a proof

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

but if we remove the last line or several of the last lines, we no longer have a proof as the outermost box does not get closed. We get a complete proof, though, by removing the last line and re-writing the assumption of the outermost box as a premise:

1	$p \wedge q \to r$	premise
2	p	premise
3	q	assumption
4	$p \wedge q$	$\wedge$i 2, 3
5	r	$\to$e 1, 4
6	$q \to r$	$\to$i 3–5

This is a proof of the sequent $p \wedge q \to r,\ p \vdash p \to r$. The induction hypothesis then ensures that $p \wedge q \to r,\ p \vDash p \to r$ holds. But then we can also reason that $p \wedge q \to r \vDash p \to (q \to r)$ holds as well – why?

Let's proceed with our proof by induction. We assume $M(k')$ for each $k' < k$ and we try to prove $M(k)$.

Base case: a one-line proof. If the proof has length 1 ($k = 1$), then it must be of the form

1	ϕ	premise

since all other rules involve more than one line. This is the case when $n = 1$ and ϕ_1 and ψ equal ϕ, i.e. we are dealing with the sequent $\phi \vdash \phi$. Of course, since ϕ evaluates to T so does ϕ. Thus, $\phi \vDash \phi$ holds as claimed.

Course-of-values inductive step: Let us assume that the proof of the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ has length k and that the statement we want to prove is true for all numbers less than k. Our proof has the following structure:

$$\begin{array}{ll}
1 & \phi_1 \text{ premise} \\
2 & \phi_2 \text{ premise} \\
 & \vdots \\
n & \phi_n \text{ premise} \\
 & \vdots \\
k & \psi \text{ justification}
\end{array}$$

There are two things we don't know at this point. First, what is happening in between those dots? Second, what was the last rule applied, i.e. what is the justification of the last line? The first uncertainty is of no concern; this is where mathematical induction demonstrates its power. The second lack of knowledge is where all the work sits. In this generality, there is simply no way of knowing which rule was applied last, so we need to consider all such rules in turn.

1. Let us suppose that this last rule is $\wedge$i. Then we know that ψ is of the form $\psi_1 \wedge \psi_2$ and the justification in line k refers to two lines further up which have ψ_1, respectively ψ_2, as their conclusions. Suppose that these lines are k_1 and k_2. Since k_1 and k_2 are smaller than k, we see that there exist proofs of the sequents $\phi_1, \phi_2, \dots, \phi_n \vdash \psi_1$ and $\phi_1, \phi_2, \dots, \phi_n \vdash \psi_2$ with length *less than* k – just take the first k_1, respectively k_2, lines of our original proof. Using the induction hypothesis, we conclude that $\phi_1, \phi_2, \dots, \phi_n \vDash \psi_1$ and $\phi_1, \phi_2, \dots, \phi_n \vDash \psi_2$ holds. But these two relations imply that $\phi_1, \phi_2, \dots, \phi_n \vDash \psi_1 \wedge \psi_2$ holds as well – why?
2. If ψ has been shown using the rule $\vee$e, then we must have proved, assumed or given as a premise some formula $\eta_1 \vee \eta_2$ in some line k' with $k' < k$, which was referred to via $\vee$e in the justification of line k. Thus, we have a shorter proof of the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \eta_1 \vee \eta_2$ within that proof, obtained by turning all assumptions of boxes that are open at line k' into premises. In a similar way we obtain proofs of the sequents $\phi_1, \phi_2, \dots, \phi_n, \eta_1 \vdash \psi$ and $\phi_1, \phi_2, \dots, \phi_n, \eta_2 \vdash \psi$ from the case analysis of $\vee$e. By our induction hypothesis, we conclude that the relations $\phi_1, \phi_2, \dots, \phi_n \vDash \eta_1 \vee \eta_2$, $\phi_1, \phi_2, \dots, \phi_n, \eta_1 \vDash \psi$ and $\phi_1, \phi_2, \dots, \phi_n, \eta_2 \vDash \psi$ hold. But together these three relations then force that $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds as well – why?
3. You can guess by now that the rest of the argument checks each possible proof rule in turn and ultimately boils down to verifying that our natural deduction

rules behave semantically in the same way as their corresponding truth tables evaluate. We leave the details as an exercise. □

The soundness of propositional logic is useful in ensuring the *non-existence* of a proof for a given sequent. Let's say you try to prove that $\phi_1, \phi_2, \dots, \phi_2 \vdash \psi$ is valid, but that your best efforts won't succeed. How could you be sure that no such proof can be found? After all, it might just be that you can't find a proof even though there is one. It suffices to find a valuation in which ϕ_i evaluate to `T` whereas ψ evaluates to `F`. Then, by definition of $\vDash$, we don't have $\phi_1, \phi_2, \dots, \phi_2 \vDash \psi$. Using soundness, this means that $\phi_1, \phi_2, \dots, \phi_2 \vdash \psi$ cannot be valid. Therefore, this sequent does not have a proof. You will practice this method in the exercises.

1.4.4 Completeness of propositional logic

In this subsection, we hope to convince you that the natural deduction rules of propositional logic are *complete*: whenever $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds, then there exists a natural deduction proof for the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$. Combined with the soundness result of the previous subsection, we then obtain

$$\phi_1, \phi_2, \dots, \phi_n \vdash \psi \text{ is valid} \quad \text{iff} \quad \phi_1, \phi_2, \dots, \phi_n \vDash \psi \text{ holds.}$$

This gives you a certain freedom regarding which method you prefer to use. Often it is much easier to show one of these two relationships (although neither of the two is universally better, or easier, to establish). The first method involves a *proof search*, upon which the *logic programming* paradigm is based. The second method typically forces you to compute a truth table which is exponential in the size of occurring propositional atoms. Both methods are intractable in general but particular instances of formulas often respond differently to treatment under these two methods.

The remainder of this section is concerned with an argument saying that if $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds, then $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ is valid. Assuming that $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds, the argument proceeds in three steps:

Step 1: We show that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to (\dots (\phi_n \to \psi) \dots)))$ holds.
Step 2: We show that $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\dots (\phi_n \to \psi) \dots)))$ is valid.
Step 3: Finally, we show that $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ is valid.

The first and third steps are quite easy; all the real work is done in the second one.

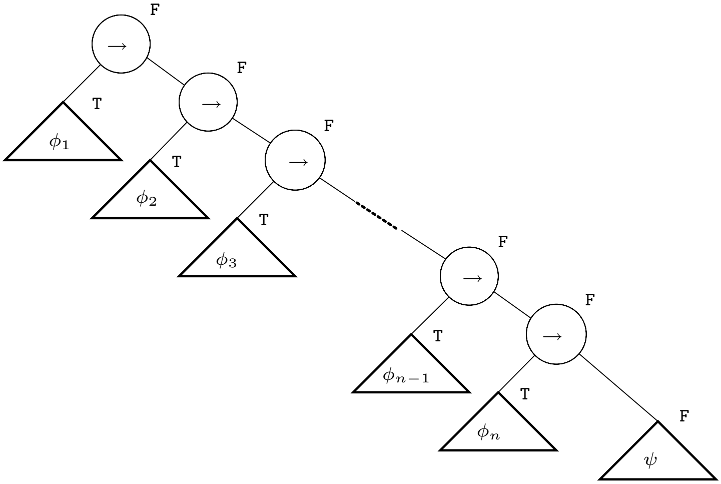


Figure 1.11. The only way this parse tree can evaluate to F. We represent parse trees for ϕ_1, ϕ_2, . . . , ϕ_n as triangles as their internal structure does not concern us here.

Step 1:

Definition 1.36 A formula of propositional logic ϕ is called a *tautology* iff it evaluates to T under all its valuations, i.e. iff $\vDash \phi$.

Supposing that $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds, let us verify that $\phi_1 \to (\phi_2 \to (\phi_3 \to (\dots (\phi_n \to \psi) \dots)))$ is indeed a tautology. Since the latter formula is a nested implication, it can evaluate to F only if all ϕ_1, ϕ_2, . . , ϕ_n evaluate to T *and* ψ evaluates to F; see its parse tree in Figure 1.11. But this contradicts the fact that $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds. Thus, $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to (\dots (\phi_n \to \psi) \dots)))$ holds.

Step 2:

Theorem 1.37 *If $\vDash \eta$ holds, then $\vdash \eta$ is valid. In other words, if η is a tautology, then η is a theorem.*

This step is the hard one. Assume that $\vDash \eta$ holds. Given that η contains n distinct propositional atoms $p_1, p_2, \dots, p_n$ we know that η evaluates to T for all 2^n lines in its truth table. (Each line lists a valuation of η.) How can we use this information to construct a proof for η? In some cases this can be done quite easily by taking a very good look at the concrete structure of η. But here we somehow have to come up with a *uniform* way of building such a proof. The key insight is to ‘encode’ each line in the truth table of η

as a sequent. Then we construct proofs for these 2^n sequents and assemble them into a proof of η.

Proposition 1.38 *Let ϕ be a formula such that $p_1, p_2, \dots, p_n$ are its only propositional atoms. Let l be any line number in ϕ's truth table. For all $1 \le i \le n$ let $\hat{p}_i$ be p_i if the entry in line l of p_i is* T*, otherwise $\hat{p}_i$ is $\neg p_i$. Then we have*

1. *$\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \phi$ is provable if the entry for ϕ in line l is* T
2. *$\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \neg\phi$ is provable if the entry for ϕ in line l is* F

PROOF: This proof is done by structural induction on the formula ϕ, that is, mathematical induction on the height of the parse tree of ϕ.

1. If ϕ is a propositional atom p, we need to show that $p \vdash p$ and $\neg p \vdash \neg p$. These have one-line proofs.
2. If ϕ is of the form $\neg\phi_1$ we again have two cases to consider. First, assume that ϕ evaluates to T. In this case ϕ_1 evaluates to F. Note that ϕ_1 has the same atomic propositions as ϕ. We may use the induction hypothesis on ϕ_1 to conclude that $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \neg\phi_1$; but $\neg\phi_1$ is just ϕ, so we are done.
   Second, if ϕ evaluates to F, then ϕ_1 evaluates to T and we get $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \phi_1$ by induction. Using the rule $\neg\neg$i, we may extend the proof of $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \phi_1$ to one for $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \neg\neg\phi_1$; but $\neg\neg\phi_1$ is just $\neg\phi$, so again we are done.

The remaining cases all deal with two subformulas: ϕ equals $\phi_1 \circ \phi_2$, where $\circ$ is $\rightarrow$, $\wedge$ or $\vee$. In all these cases let $q_1, \dots, q_l$ be the propositional atoms of ϕ_1 and $r_1, \dots, r_k$ be the propositional atoms of ϕ_2. Then we certainly have $\{q_1, \dots, q_l\} \cup \{r_1, \dots, r_k\} = \{p_1, \dots, p_n\}$. Therefore, whenever $\hat{q}_1, \dots, \hat{q}_l \vdash \psi_1$ and $\hat{r}_1, \dots, \hat{r}_k \vdash \psi_2$ are valid so is $\hat{p}_1, \dots, \hat{p}_n \vdash \psi_1 \wedge \psi_2$ using the rule $\wedge$i. In this way, we can use our induction hypothesis and only owe proofs that the conjunctions we conclude allow us to prove the desired conclusion for ϕ or $\neg\phi$ as the case may be.

3. To wit, let ϕ be $\phi_1 \rightarrow \phi_2$. If ϕ evaluates to F, then we know that ϕ_1 evaluates to T and ϕ_2 to F. Using our induction hypothesis, we have $\hat{q}_1, \dots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \dots, \hat{r}_k \vdash \neg\phi_2$, so $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ follows. We need to show $\hat{p}_1, \dots, \hat{p}_n \vdash \neg(\phi_1 \rightarrow \phi_2)$; but using $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$, this amounts to proving the sequent $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \rightarrow \phi_2)$, which we leave as an exercise.
   If ϕ evaluates to T, then we have three cases. First, if ϕ_1 evaluates to F and ϕ_2 to F, then we get, by our induction hypothesis, that $\hat{q}_1, \dots, \hat{q}_l \vdash \neg\phi_1$ and $\hat{r}_1, \dots, \hat{r}_k \vdash \neg\phi_2$, so $\hat{p}_1, \dots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ follows. Again, we need only to show the sequent $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \rightarrow \phi_2$, which we leave as an exercise. Second, if ϕ_1 evaluates to F and ϕ_2 to T, we use our induction hypothesis to arrive at

$\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ and have to prove $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \rightarrow \phi_2$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to T, we arrive at $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, using our induction hypothesis, and need to prove $\phi_1 \wedge \phi_2 \vdash \phi_1 \rightarrow \phi_2$, which we leave as an exercise as well.

4. If ϕ is of the form $\phi_1 \wedge \phi_2$, we are again dealing with four cases in total. First, if ϕ_1 and ϕ_2 evaluate to T, we get $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \phi_2$ by our induction hypothesis, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ follows. Second, if ϕ_1 evaluates to F and ϕ_2 to T, then we get $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ using our induction hypothesis and the rule $\wedge$i as above and we need to prove $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to F, then our induction hypothesis and the rule $\wedge$i let us infer that $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$; so we are left with proving $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise. Fourth, if ϕ_1 evaluates to T and ϕ_2 to F, we obtain $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ by our induction hypothesis and we have to show $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise.
5. Finally, if ϕ is a disjunction $\phi_1 \vee \phi_2$, we again have four cases. First, if ϕ_1 and ϕ_2 evaluate to F, then our induction hypothesis and the rule $\wedge$i give us $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ and we have to show $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \vee \phi_2)$, which we leave as an exercise. Second, if ϕ_1 and ϕ_2 evaluate to T, then we obtain $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, by our induction hypothesis, and we need a proof for $\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. Third, if ϕ_1 evaluates to F and ϕ_2 to T, then we arrive at $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$, using our induction hypothesis, and need to establish $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. Fourth, if ϕ_1 evaluates to T and ϕ_2 to F, then $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ results from our induction hypothesis and all we need is a proof for $\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. □

We apply this technique to the formula $\vDash \phi_1 \rightarrow (\phi_2 \rightarrow (\phi_3 \rightarrow (\ldots (\phi_n \rightarrow \psi) \ldots)))$. Since it is a tautology it evaluates to T in all 2^n lines of its truth table; thus, the proposition above gives us 2^n many proofs of $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \eta$, one for each of the cases that $\hat{p}_i$ is p_i or $\neg p_i$. Our job now is to assemble all these proofs into a single proof for η which does not use any premises. We illustrate how to do this for an example, the tautology $p \wedge q \rightarrow p$.

The formula $p \wedge q \rightarrow p$ has two propositional atoms p and q. By the proposition above, we are guaranteed to have a proof for each of the four sequents

$$\begin{aligned} p, q &\vdash p \wedge q \rightarrow p \\ \neg p, q &\vdash p \wedge q \rightarrow p \\ p, \neg q &\vdash p \wedge q \rightarrow p \\ \neg p, \neg q &\vdash p \wedge q \rightarrow p. \end{aligned}$$

Ultimately, we want to prove $p \wedge q \rightarrow p$ by appealing to the four proofs of the sequents above. Thus, we somehow need to get rid of the premises on the left-hand sides of these four sequents. This is the place where we rely on the law of the excluded middle which states $r \lor \neg r$, for any r. We use LEM for all propositional atoms (here p and q) and then we separately assume all the four cases, by using $\lor$e. That way we can invoke all four proofs of the sequents above and use the rule $\lor$e repeatedly until we have got rid of all our premises. We spell out the combination of these four phases schematically:

1	$p \lor \neg p$							LEM
2	p			ass	$\neg p$			ass
3	$q \lor \neg q$			LEM	$q \lor \neg q$			LEM
4	q	ass	$\neg q$	ass	q	ass	$\neg q$	ass
5	$\vdots$		$\vdots$		$\vdots$		$\vdots$	
6								
7	$p \land q \to p$		$p \land q \to p$		$p \land q \to p$		$p \land q \to p$	
8	$p \land q \to p$			$\lor$e	$p \land q \to p$			$\lor$e
9	$p \land q \to p$							$\lor$e

As soon as you understand how this particular example works, you will also realise that it will work for an arbitrary tautology with n distinct atoms. Of course, it seems ridiculous to prove $p \land q \to p$ using a proof that is this long. But remember that this illustrates a *uniform* method that constructs a proof for every tautology η, no matter how complicated it is.

Step 3: Finally, we need to find a proof for $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$. Take the proof for $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\dots(\phi_n \to \psi)\dots)))$ given by step 2 and augment its proof by introducing $\phi_1, \phi_2, \dots, \phi_n$ as premises. Then apply $\to$e n times on each of these premises (starting with ϕ_1, continuing with ϕ_2 etc.). Thus, we arrive at the conclusion ψ which gives us a proof for the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$.

Corollary 1.39 (Soundness and Completeness) *Let* $\phi_1, \phi_2, \dots, \phi_n, \psi$ *be formulas of propositional logic. Then* $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ *is holds iff the sequent* $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ *is valid.*

1.5 Normal forms

In the last section, we showed that our proof system for propositional logic is sound and complete for the truth-table semantics of formulas in Figure 1.6.

Soundness means that whatever we prove is going to be a true fact, based on the truth-table semantics. In the exercises, we apply this to show that a sequent does not have a proof: simply show that $\phi_1, \phi_2, \ldots, \phi_2$ does not semantically entail ψ; then soundness implies that the sequent $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ does not have a proof. Completeness comprised a much more powerful statement: no matter what (semantically) valid sequents there are, they all have syntactic proofs in the proof system of natural deduction. This tight correspondence allows us to freely switch between working with the notion of proofs ($\vdash$) and that of semantic entailment ($\vDash$).

Using natural deduction to decide the validity of instances of $\vdash$ is only one of many possibilities. In Exercise 1.2.6 we sketch a non-linear, tree-like, notion of proofs for sequents. Likewise, checking an instance of $\vDash$ by applying Definition 1.34 literally is only one of many ways of deciding whether $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds. We now investigate various alternatives for deciding $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ which are based on transforming these formulas syntactically into 'equivalent' ones upon which we can then settle the matter by purely syntactic or algorithmic means. This requires that we first clarify what exactly we mean by equivalent formulas.

1.5.1 Semantic equivalence, satisfiability and validity

Two formulas ϕ and ψ are said to be equivalent if they have the same 'meaning.' This suggestion is vague and needs to be refined. For example, $p \to q$ and $\neg p \vee q$ have the same truth table; all four combinations of T and F for p and q return the same result. 'Coincidence of truth tables' is not good enough for what we have in mind, for what about the formulas $p \wedge q \to p$ and $r \vee \neg r$? At first glance, they have little in common, having different atomic formulas and different connectives. Moreover, the truth table for $p \wedge q \to p$ is four lines long, whereas the one for $r \vee \neg r$ consists of only two lines. However, both formulas are always true. This suggests that we define the equivalence of formulas ϕ and ψ via $\vDash$: if ϕ semantically entails ψ and vice versa, then these formulas should be the same as far as our truth-table semantics is concerned.

Definition 1.40 Let ϕ and ψ be formulas of propositional logic. We say that ϕ and ψ are *semantically equivalent* iff $\phi \vDash \psi$ and $\psi \vDash \phi$ hold. In that case we write $\phi \equiv \psi$. Further, we call ϕ *valid* if $\vDash \phi$ holds.

Note that we could also have defined $\phi \equiv \psi$ to mean that $\vDash (\phi \to \psi) \wedge (\psi \to \phi)$ holds; it amounts to the same concept. Indeed, because of soundness and completeness, semantic equivalence is identical to *provable equivalence*

(Definition 1.25). Examples of equivalent formulas are

$$\begin{aligned} p \to q &\equiv \neg q \to \neg p \\ p \to q &\equiv \neg p \lor q \\ p \land q \to p &\equiv r \lor \neg r \\ p \land q \to r &\equiv p \to (q \to r). \end{aligned}$$

Recall that a formula η is called a tautology if $\vDash \eta$ holds, so the tautologies are exactly the valid formulas. The following lemma says that any decision procedure for tautologies is in fact a decision procedure for the validity of sequents as well.

Lemma 1.41 *Given formulas $\phi_1, \phi_2, \dots, \phi_n$ and ψ of propositional logic, $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds iff $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \dots \to (\phi_n \to \psi)))$ holds.*

PROOF: First, suppose that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \dots \to (\phi_n \to \psi)))$ holds. If $\phi_1, \phi_2, \dots, \phi_n$ are all true under some valuation, then ψ has to be true as well for that same valuation. Otherwise, $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \dots \to (\phi_n \to \psi)))$ would not hold (compare this with Figure 1.11). Second, if $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds, we have already shown that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \dots \to (\phi_n \to \psi)))$ follows in step 1 of our completeness proof. □

For our current purposes, we want to transform formulas into ones which don't contain $\to$ at all and the occurrences of $\land$ and $\lor$ are confined to separate layers such that validity checks are easy. This is being done by

1. using the equivalence $\phi \to \psi \equiv \neg\phi \lor \psi$ to remove all occurrences of $\to$ from a formula and
2. by specifying an algorithm that takes a formula without any $\to$ into a *normal form* (still without $\to$) for which checking validity is easy.

Naturally, we have to specify which forms of formulas we think of as being 'normal.' Again, there are many such notions, but in this text we study only two important ones.

Definition 1.42 A *literal* L is either an atom p or the negation of an atom $\neg p$. A formula C is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where each clause D is a disjunction of literals:

$$\begin{aligned} L &::= p \mid \neg p \\ D &::= L \mid L \lor D \\ C &::= D \mid D \land C. \end{aligned} \tag{1.6}$$

Examples of formulas in conjunctive normal form are

$$(i) \quad (\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q \qquad (ii) \quad (p \vee r) \wedge (\neg p \vee r) \wedge (p \vee \neg r).$$

In the first case, there are three clauses of type D: $\neg q \vee p \vee r$, $\neg p \vee r$, and q – which is a literal promoted to a clause by the first rule of clauses in (1.6). Notice how we made implicit use of the associativity laws for $\wedge$ and $\vee$, saying that $\phi \vee (\psi \vee \eta) \equiv (\phi \vee \psi) \vee \eta$ and $\phi \wedge (\psi \wedge \eta) \equiv (\phi \wedge \psi) \wedge \eta$, since we omitted some parentheses. The formula $(\neg(q \vee p) \vee r) \wedge (q \vee r)$ is not in CNF since $q \vee p$ is not a literal.

Why do we care at all about formulas ϕ in CNF? One of the reasons for their usefulness is that they allow easy checks of validity which otherwise take times exponential in the number of atoms. For example, consider the formula in CNF from above: $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$. The semantic entailment $\vDash (\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ holds iff all three relations

$$\vDash \neg q \vee p \vee r \qquad \vDash \neg p \vee r \qquad \vDash q$$

hold, by the semantics of $\wedge$. But since all of these formulas are disjunctions of literals, or literals, we can settle the matter as follows.

Lemma 1.43 *A disjunction of literals $L_1 \vee L_2 \vee \cdots \vee L_m$ is valid iff there are $1 \leq i, j \leq m$ such that L_i is $\neg L_j$.*

PROOF: If L_i equals $\neg L_j$, then $L_1 \vee L_2 \vee \cdots \vee L_m$ evaluates to T for all valuations. For example, the disjunct $p \vee q \vee r \vee \neg q$ can never be made false.

To see that the converse holds as well, assume that no literal L_k has a matching negation in $L_1 \vee L_2 \vee \cdots \vee L_m$. Then, for each k with $1 \leq k \leq n$, we assign F to L_k, if L_k is an atom; or T, if L_k is the negation of an atom. For example, the disjunct $\neg q \vee p \vee r$ can be made false by assigning F to p and r and T to q. □

Hence, we have an easy and fast check for the validity of $\vDash \phi$, provided that ϕ is in CNF; inspect all conjuncts ψ_k of ϕ and search for atoms in ψ_k such that ψ_k also contains their negation. If such a match is found for all conjuncts, we have $\vDash \phi$. Otherwise (= some conjunct contains no pair L_i and $\neg L_i$), ϕ is not valid by the lemma above. Thus, the formula $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ above is not valid. Note that the matching literal has to be found in the same conjunct ψ_k. Since there is no free lunch in this universe, we can expect that the computation of a formula ϕ' in CNF, which is equivalent to a given formula ϕ, is a costly worst-case operation.

Before we study how to compute equivalent conjunctive normal forms, we introduce another semantic concept closely related to that of validity.

Definition 1.44 Given a formula ϕ in propositional logic, we say that ϕ is *satisfiable* if it has a valuation in which is evaluates to T.

For example, the formula $p \vee q \to p$ is satisfiable since it computes T if we assign T to p. Clearly, $p \vee q \to p$ is not valid. Thus, satisfiability is a weaker concept since every valid formula is by definition also satisfiable but not vice versa. However, these two notions are just mirror images of each other, the mirror being negation.

Proposition 1.45 *Let ϕ be a formula of propositional logic. Then ϕ is satisfiable iff $\neg\phi$ is not valid.*

Proof: First, assume that ϕ is satisfiable. By definition, there exists a valuation of ϕ in which ϕ evaluates to T; but that means that $\neg\phi$ evaluates to F for that same valuation. Thus, $\neg\phi$ cannot be valid.

Second, assume that $\neg\phi$ is not valid. Then there must be a valuation of $\neg\phi$ in which $\neg\phi$ evaluates to F. Thus, ϕ evaluates to T and is therefore satisfiable. (Note that the valuations of ϕ are exactly the valuations of $\neg\phi$.) □

This result is extremely useful since it essentially says that we need provide a decision procedure for only one of these concepts. For example, let's say that we have a procedure P for deciding whether any ϕ is valid. We obtain a decision procedure for satisfiability simply by asking P whether $\neg\phi$ is valid. If it is, ϕ is not satisfiable; otherwise ϕ is satisfiable. Similarly, we may transform any decision procedure for satisfiability into one for validity. We will encounter both kinds of procedures in this text.

There is one scenario in which computing an equivalent formula in CNF is really easy; namely, when someone else has already done the work of writing down a full truth table for ϕ. For example, take the truth table of $(p \to \neg q) \to (q \vee \neg p)$ in Figure 1.8 (page 40). For each line where $(p \to \neg q) \to (q \vee \neg p)$ computes F we now construct a disjunction of literals. Since there is only one such line, we have only one conjunct ψ_1. That conjunct is now obtained by a disjunction of literals, where we include literals $\neg p$ and q. Note that the literals are just the syntactic opposites of the truth values in that line: here p is T and q is F. The resulting formula in CNF is thus $\neg p \vee q$ which is readily seen to be in CNF and to be equivalent to $(p \to \neg q) \to (q \vee \neg p)$.

Why does this always work for any formula ϕ? Well, the constructed formula will be false iff at least one of its conjuncts ψ_i will be false. This means that all the disjuncts in such a ψ_i must be F. Using the de Morgan rule $\neg\phi_1 \lor \neg\phi_2 \lor \cdots \lor \neg\phi_n \equiv \neg(\phi_1 \land \phi_2 \land \cdots \land \phi_n)$, we infer that the conjunction of the syntactic opposites of those literals must be true. Thus, ϕ and the constructed formula have the same truth table.

Consider another example, in which ϕ is given by the truth table:

p	q	r	ϕ
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

Note that this table is really just a specification of ϕ; it does not tell us what ϕ looks like syntactically, but it does tells us how it ought to 'behave.' Since this truth table has four entries which compute F, we construct four conjuncts ψ_i $(1 \leq i \leq 4)$. We read the ψ_i off that table by listing the disjunction of all atoms, where we negate those atoms which are true in those lines:

$$\psi_1 \stackrel{\text{def}}{=} \neg p \lor \neg q \lor r \quad \text{(line 2)} \qquad \psi_2 \stackrel{\text{def}}{=} p \lor \neg q \lor \neg r \quad \text{(line 5)}$$

$$\psi_3 \stackrel{\text{def}}{=} p \lor \neg q \lor r \quad \text{etc} \qquad \psi_4 \stackrel{\text{def}}{=} p \lor q \lor \neg r.$$

The resulting ϕ in CNF is therefore

$$(\neg p \lor \neg q \lor r) \land (p \lor \neg q \lor \neg r) \land (p \lor \neg q \lor r) \land (p \lor q \lor \neg r).$$

If we don't have a full truth table at our disposal, but do know the structure of ϕ, then we would like to compute a version of ϕ in CNF. It should be clear by now that a full truth table of ϕ and an equivalent formula in CNF are pretty much the same thing as far as questions about validity are concerned – although the formula in CNF may be much more compact.

1.5.2 Conjunctive normal forms and validity

We have already seen the benefits of conjunctive normal forms in that they allow for a fast and easy syntactic test of validity. Therefore, one wonders whether any formula can be transformed into an *equivalent* formula in CNF. We now develop an algorithm achieving just that. Note that, by Definition 1.40, a formula is valid iff any of its equivalent formulas is valid. We reduce the problem of determining whether *any* ϕ is valid to the problem of computing an equivalent $\psi \equiv \phi$ such that ψ is in CNF and checking, via Lemma 1.43, whether ψ is valid.

Before we sketch such a procedure, we make some general remarks about its possibilities and its realisability constraints. First of all, there could be more or less efficient ways of computing such normal forms. But even more so, there could be many possible correct outputs, for $\psi_1 \equiv \phi$ and $\psi_2 \equiv \phi$ do not generally imply that ψ_1 is the same as ψ_2, even if ψ_1 and ψ_2 are in CNF. For example, take $\phi \stackrel{\text{def}}{=} p$, $\psi_1 \stackrel{\text{def}}{=} p$ and $\psi_2 \stackrel{\text{def}}{=} p \land (p \lor q)$; then convince yourself that $\phi \equiv \psi_2$ holds. Having this ambiguity of equivalent conjunctive normal forms, the computation of a CNF for ϕ with minimal 'cost' (where 'cost' could for example be the number of conjuncts, or the height of ϕ's parse tree) becomes a very important practical problem, an issue persued in Chapter 6. Right now, we are content with stating a *deterministic* algorithm which always computes the same output CNF for a given input ϕ.

This algorithm, called `CNF`, should satisfy the following requirements:

(1) `CNF` terminates for all formulas of propositional logic as input;
(2) for each such input, `CNF` outputs an equivalent formula; and
(3) all output computed by `CNF` is in CNF.

If a call of `CNF` with a formula ϕ of propositional logic as input terminates, which is enforced by (1), then (2) ensures that $\psi \equiv \phi$ holds for the output ψ. Thus, (3) guarantees that ψ is an equivalent CNF of ϕ. So ϕ is valid iff ψ is valid; and checking the latter is easy relative to the length of ψ.

What kind of strategy should `CNF` employ? It will have to function correctly for all, i.e. infinitely many, formulas of propositional logic. This strongly suggests to write a procedure that computes a CNF by structural induction on the formula ϕ. For example, if ϕ is of the form $\phi_1 \land \phi_2$, we may simply compute conjunctive normal forms η_i for ϕ_i $(i = 1, 2)$, whereupon $\eta_1 \land \eta_2$ is a conjunctive normal form which is equivalent to ϕ *provided that* $\eta_i \equiv \phi_i$ $(i = 1, 2)$. This strategy also suggests to use proof by structural induction on ϕ to prove that `CNF` meets the requirements (1–3) stated above.

Given a formula ϕ as input, we first do some *preprocessing*. Initially, we translate away all implications in ϕ by replacing all subformulas of the form $\psi \to \eta$ by $\neg\psi \lor \eta$. This is done by a procedure called `IMPL_FREE`. Note that this procedure has to be recursive, for there might be implications in ψ or η as well.

The application of `IMPL_FREE` might introduce double negations into the output formula. More importantly, negations whose scopes are non-atomic formulas might still be present. For example, the formula $p \land \neg(p \land q)$ has such a negation with $p \land q$ as its scope. Essentially, the question is whether one can efficiently compute a CNF for $\neg\phi$ from a CNF for ϕ. Since *nobody* seems to know the answer, we circumvent the question by translating $\neg\phi$

into an equivalent formula that contains only negations of atoms. Formulas which only negate atoms are said to be in *negation normal form* (NNF). We spell out such a procedure, NNF, in detail later on. The key to its specification for implication-free formulas lies in the de Morgan rules. The second phase of the preprocessing, therefore, calls NNF with the implication-free output of IMPL_FREE to obtain an equivalent formula in NNF.

After all this preprocessing, we obtain a formula ϕ' which is the result of the call NNF (IMPL_FREE (ϕ)). Note that $\phi' \equiv \phi$ since both algorithms only transform formulas into equivalent ones. Since ϕ' contains no occurrences of $\rightarrow$ and since only atoms in ϕ' are negated, we may program CNF by an analysis of only *three* cases: literals, conjunctions and disjunctions.

- If ϕ is a literal, it is by definition in CNF and so CNF outputs ϕ.
- If ϕ equals $\phi_1 \wedge \phi_2$, we call CNF recursively on each ϕ_i to get the respective output η_i and return the CNF $\eta_1 \wedge \eta_2$ as output for input ϕ.
- If ϕ equals $\phi_1 \vee \phi_2$, we again call CNF recursively on each ϕ_i to get the respective output η_i; but this time we must not simply return $\eta_1 \vee \eta_2$ since that formula is certainly *not* in CNF, unless η_1 and η_2 happen to be literals.

So how can we complete the program in the last case? Well, we may resort to the distributivity laws, which entitle us to translate any disjunction of conjunctions into a conjunction of disjunctions. However, for this to result in a CNF, we need to make certain that those disjunctions generated contain only literals. We apply a strategy for using distributivity based on matching patterns in $\phi_1 \vee \phi_2$. This results in an independent algorithm called DISTR which will do all that work for us. Thus, we simply call DISTR with the pair (η_1, η_2) as input and pass along its result.

Assuming that we already have written code for IMPL_FREE, NNF and DISTR, we may now write pseudo code for CNF:

```
function CNF ($\phi$):
/* precondition: $\phi$ implication free and in NNF */
/* postcondition: CNF ($\phi$) computes an equivalent CNF for $\phi$ */
begin function
  case
    $\phi$ is a literal: return $\phi$
    $\phi$ is $\phi_1 \wedge \phi_2$: return CNF ($\phi_1$) $\wedge$ CNF ($\phi_2$)
    $\phi$ is $\phi_1 \vee \phi_2$: return DISTR (CNF ($\phi_1$), CNF ($\phi_2$))
  end case
end function
```

Notice how the calling of DISTR is done with the computed conjunctive normal forms of ϕ_1 and ϕ_2. The routine DISTR has η_1 and η_2 as input parameters and does a case analysis on whether these inputs are conjunctions. What should DISTR do if none of its input formulas is such a conjunction? Well, since we are calling DISTR for inputs η_1 and η_2 which are in CNF, this can only mean that η_1 and η_2 are literals, or disjunctions of literals. Thus, $\eta_1 \vee \eta_2$ is in CNF.

Otherwise, at least one of the formulas η_1 and η_2 is a conjunction. Since one conjunction suffices for simplifying the problem, we have to decide which conjunct we want to transform if *both* formulas are conjunctions. That way we maintain that our algorithm CNF is deterministic. So let us suppose that η_1 is of the form $\eta_{11} \wedge \eta_{12}$. Then the distributive law says that $\eta_1 \vee \eta_2 \equiv (\eta_{11} \vee \eta_2) \wedge (\eta_{12} \vee \eta_2)$. Since all participating formulas η_{11}, η_{12} and η_2 are in CNF, we may call DISTR again for the pairs (η_{11}, η_2) and (η_{12}, η_2), and then simply form their conjunction. This is the key insight for writing the function DISTR.

The case when η_2 is a conjunction is symmetric and the structure of the recursive call of DISTR is then dictated by the equivalence $\eta_1 \vee \eta_2 \equiv (\eta_1 \vee \eta_{21}) \wedge (\eta_1 \vee \eta_{22})$, where $\eta_2 = \eta_{21} \wedge \eta_{22}$:

```
function DISTR ($\eta_1, \eta_2$):
/* precondition: $\eta_1$ and $\eta_2$ are in CNF */
/* postcondition: DISTR ($\eta_1, \eta_2$) computes a CNF for $\eta_1 \vee \eta_2$ */
begin function
  case
    $\eta_1$ is $\eta_{11} \wedge \eta_{12}$: return DISTR ($\eta_{11}, \eta_2$) $\wedge$ DISTR ($\eta_{12}, \eta_2$)
    $\eta_2$ is $\eta_{21} \wedge \eta_{22}$: return DISTR ($\eta_1, \eta_{21}$) $\wedge$ DISTR ($\eta_1, \eta_{22}$)
    otherwise (= no conjunctions): return $\eta_1 \vee \eta_2$
  end case
end function
```

Notice how the three clauses are exhausting all possibilities. Furthermore, the first and second cases overlap if η_1 and η_2 are both conjunctions. It is then our understanding that this code will inspect the clauses of a case statement from the top to the bottom clause. Thus, the first clause would apply.

Having specified the routines CNF and DISTR, this leaves us with the task of writing the functions IMPL_FREE and NNF. We delegate the design of IMPL_FREE to the exercises. The function NNF has to transform any implication-free formula into an equivalent one in negation normal form. Four examples of formulas in NNF are

$$\begin{array}{ll} p & \neg p \\ \neg p \wedge (p \wedge q) & \neg p \wedge (p \rightarrow q), \end{array}$$

although we won't have to deal with a formula of the last kind since → won't occur. Examples of formulas which are not in NNF are $\neg\neg p$ and $\neg(p \wedge q)$.

Again, we program NNF recursively by a case analysis over the structure of the input formula ϕ. The last two examples already suggest a solution for two of these clauses. In order to compute a NNF of $\neg\neg\phi$, we simply compute a NNF of ϕ. This is a sound strategy since ϕ and $\neg\neg\phi$ are semantically equivalent. If ϕ equals $\neg(\phi_1 \wedge \phi_2)$, we use the de Morgan rule $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2$ as a recipe for how NNF should call itself recursively in that case. Dually, the case of ϕ being $\neg(\phi_1 \vee \phi_2)$ appeals to the other de Morgan rule $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$ and, if ϕ is a conjunction or disjunction, we simply let NNF pass control to those subformulas. Clearly, all literals are in NNF. The resulting code for NNF is thus

```
function NNF ($\phi$):
/* precondition: $\phi$ is implication free */
/* postcondition: NNF ($\phi$) computes a NNF for $\phi$ */
begin function
  case
    $\phi$ is a literal: return $\phi$
    $\phi$ is $\neg\neg\phi_1$: return NNF ($\phi_1$)
    $\phi$ is $\phi_1 \wedge \phi_2$: return NNF ($\phi_1$) $\wedge$ NNF ($\phi_2$)
    $\phi$ is $\phi_1 \vee \phi_2$: return NNF ($\phi_1$) $\vee$ NNF ($\phi_2$)
    $\phi$ is $\neg(\phi_1 \wedge \phi_2)$: return NNF ($\neg\phi_1$) $\vee$ NNF ($\neg\phi_2$)
    $\phi$ is $\neg(\phi_1 \vee \phi_2)$: return NNF ($\neg\phi_1$) $\wedge$ NNF ($\neg\phi_2$)
  end case
end function
```

Notice that these cases are exhaustive due to the algorithm's precondition. Given any formula ϕ of propositional logic, we may now convert it into an equivalent CNF by calling $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi)))$. In the exercises, you are asked to show that

- all four algorithms terminate on input meeting their preconditions,
- the result of $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi)))$ is in CNF and
- that result is semantically equivalent to ϕ.

We will return to the important issue of formally proving the correctness of programs in Chapter 4.

Let us now illustrate the programs coded above on some concrete examples. We begin by computing $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\neg p \wedge q \rightarrow p \wedge (r \rightarrow q))))$. We show almost all details of this computation and you should compare this with how you would expect the code above to behave. First, we compute $\texttt{IMPL\_FREE}\ (\phi)$:

$$\begin{aligned}
\texttt{IMPL\_FREE}\,(\phi) &= \neg\texttt{IMPL\_FREE}\,(\neg p \wedge q) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \rightarrow q)) \\
&= \neg((\texttt{IMPL\_FREE}\,\neg p) \wedge (\texttt{IMPL\_FREE}\,q)) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \rightarrow q)) \\
&= \neg((\neg p) \wedge \texttt{IMPL\_FREE}\,q) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \rightarrow q)) \\
&= \neg(\neg p \wedge q) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \rightarrow q)) \\
&= \neg(\neg p \wedge q) \vee ((\texttt{IMPL\_FREE}\,p) \wedge \texttt{IMPL\_FREE}\,(r \rightarrow q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge \texttt{IMPL\_FREE}\,(r \rightarrow q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg(\texttt{IMPL\_FREE}\,r) \vee (\texttt{IMPL\_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee (\texttt{IMPL\_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}$$

Second, we compute $\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi)$:

$$\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi) &= \texttt{NNF}\ (\neg(\neg p \wedge q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= \texttt{NNF}\ (\neg(\neg p) \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (\texttt{NNF}\ (\neg\neg p)) \vee (\texttt{NNF}\ (\neg q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee (\texttt{NNF}\ (\neg q))) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee ((\texttt{NNF}\ p) \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge ((\texttt{NNF}\ (\neg r)) \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}$$

Third, we finish it off with

$$\begin{aligned}
\texttt{CNF}\ (\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi)) &= \texttt{CNF}\ ((p \lor \neg q) \lor (p \land (\neg r \lor q))) \\
&= \texttt{DISTR}\ (\texttt{CNF}\ (p \lor \neg q), \texttt{CNF}\ (p \land (\neg r \lor q))) \\
&= \texttt{DISTR}\ (p \lor \neg q, \texttt{CNF}\ (p \land (\neg r \lor q))) \\
&= \texttt{DISTR}\ (p \lor \neg q, p \land (\neg r \lor q)) \\
&= \texttt{DISTR}\ (p \lor \neg q, p) \land \texttt{DISTR}\ (p \lor \neg q, \neg r \lor q) \\
&= (p \lor \neg q \lor p) \land \texttt{DISTR}\ (p \lor \neg q, \neg r \lor q) \\
&= (p \lor \neg q \lor p) \land (p \lor \neg q \lor \neg r \lor q)\ .
\end{aligned}$$

The formula $(p \lor \neg q \lor p) \land (p \lor \neg q \lor \neg r \lor q)$ is thus the result of the call $\texttt{CNF}\ (\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi))$ and is in conjunctive normal form and equivalent to ϕ. Note that it is satisfiable (choose p to be true) but not valid (choose p to be false and q to be true); it is also equivalent to the simpler conjunctive normal form $p \lor \neg q$. Observe that our algorithm does not do such optimisations so one would need a separate optimiser running on the output. Alternatively, one might change the code of our functions to allow for such optimisations 'on the fly,' a computational overhead which could prove to be counterproductive.

You should realise that we omitted several computation steps in the subcalls $\texttt{CNF}\ (p \lor \neg q)$ and $\texttt{CNF}\ (p \land (\neg r \lor q))$. They return their input as a result since the input is already in conjunctive normal form.

As a second example, consider $\phi \stackrel{\text{def}}{=} r \to (s \to (t \land s \to r))$. We compute

$$\begin{aligned}
\texttt{IMPL\_FREE}\,(\phi) &= \neg(\texttt{IMPL\_FREE}\ r) \lor \texttt{IMPL\_FREE}\,(s \to (t \land s \to r)) \\
&= \neg r \lor \texttt{IMPL\_FREE}\,(s \to (t \land s \to r)) \\
&= \neg r \lor (\neg(\texttt{IMPL\_FREE}\,s) \lor \texttt{IMPL\_FREE}\,(t \land s \to r)) \\
&= \neg r \lor (\neg s \lor \texttt{IMPL\_FREE}\,(t \land s \to r)) \\
&= \neg r \lor (\neg s \lor (\neg(\texttt{IMPL\_FREE}\,(t \land s)) \lor \texttt{IMPL\_FREE}\,r)) \\
&= \neg r \lor (\neg s \lor (\neg((\texttt{IMPL\_FREE}\,t) \land (\texttt{IMPL\_FREE}\,s)) \lor \texttt{IMPL\_FREE}\,r)) \\
&= \neg r \lor (\neg s \lor (\neg(t \land (\texttt{IMPL\_FREE}\,s)) \lor (\texttt{IMPL\_FREE}\,r))) \\
&= \neg r \lor (\neg s \lor (\neg(t \land s)) \lor (\texttt{IMPL\_FREE}\,r)) \\
&= \neg r \lor (\neg s \lor (\neg(t \land s)) \lor r)
\end{aligned}$$

$$
\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi) &= \texttt{NNF}\ (\neg r \vee (\neg s \vee \neg(t \wedge s) \vee r))\\
&= (\texttt{NNF}\ \neg r) \vee \texttt{NNF}\ (\neg s \vee \neg(t \wedge s) \vee r)\\
&= \neg r \vee \texttt{NNF}\ (\neg s \vee \neg(t \wedge s) \vee r)\\
&= \neg r \vee (\texttt{NNF}\ (\neg s) \vee \texttt{NNF}\ (\neg(t \wedge s) \vee r))\\
&= \neg r \vee (\neg s \vee \texttt{NNF}\ (\neg(t \wedge s) \vee r))\\
&= \neg r \vee (\neg s \vee (\texttt{NNF}\ (\neg(t \wedge s)) \vee \texttt{NNF}\ r))\\
&= \neg r \vee (\neg s \vee (\texttt{NNF}\ (\neg t \vee \neg s)) \vee \texttt{NNF}\ r)\\
&= \neg r \vee (\neg s \vee ((\texttt{NNF}\ (\neg t) \vee \texttt{NNF}\ (\neg s)) \vee \texttt{NNF}\ r))\\
&= \neg r \vee (\neg s \vee ((\neg t \vee \texttt{NNF}\ (\neg s)) \vee \texttt{NNF}\ r))\\
&= \neg r \vee (\neg s \vee ((\neg t \vee \neg s) \vee \texttt{NNF}\ r))\\
&= \neg r \vee (\neg s \vee ((\neg t \vee \neg s) \vee r))
\end{aligned}
$$

where the latter is already in CNF and valid as r has a matching $\neg r$.

1.5.3 Horn clauses and satisfiability

We have already commented on the computational price we pay for transforming a propositional logic formula into an equivalent CNF. The latter class of formulas has an easy syntactic check for validity, but its test for satisfiability is very hard in general. Fortunately, there are practically important subclasses of formulas which have much more efficient ways of deciding their satisfiability. One such example is the class of *Horn formulas*; the name 'Horn' is derived from the logician A. Horn's last name. We shortly define them and give an algorithm for checking their satisfiability.

Recall that the logical constants $\bot$ ('bottom') and $\top$ ('top') denote an unsatisfiable formula, respectively, a tautology.

Definition 1.46 A *Horn formula* is a formula ϕ of propositional logic if it can be generated as an instance of H in this grammar:

$$
\begin{aligned}
P &::= \bot \ \mid\ \top \ \mid\ p\\
A &::= P \ \mid\ P \wedge A\\
C &::= A \to P\\
H &::= C \ \mid\ C \wedge H.
\end{aligned}
\tag{1.7}
$$

We call each instance of C a *Horn clause*.

Horn formulas are conjunctions of Horn clauses. A Horn clause is an implication whose assumption A is a conjunction of propositions of type P and whose conclusion is also of type P. Examples of Horn formulas are

$$(p \wedge q \wedge s \to p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$$
$$(p \wedge q \wedge s \to \bot) \wedge (q \wedge r \to p) \wedge (\top \to s)$$
$$(p_2 \wedge p_3 \wedge p_5 \to p_{13}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \to \bot).$$

Examples of formulas which are *not* Horn formulas are

$$(p \wedge q \wedge s \to \neg p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$$
$$(p \wedge q \wedge s \to \bot) \wedge (\neg q \wedge r \to p) \wedge (\top \to s)$$
$$(p_2 \wedge p_3 \wedge p_5 \to p_{13} \wedge p_{27}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \to \bot)$$
$$(p_2 \wedge p_3 \wedge p_5 \to p_{13} \wedge p_{27}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \vee \bot).$$

The first formula is not a Horn formula since $\neg p$, the conclusion of the implication of the first conjunct, is not of type P. The second formula does not qualify since the premise of the implication of the second conjunct, $\neg q \wedge r$, is not a conjunction of atoms, $\bot$, or $\top$. The third formula is not a Horn formula since the conclusion of the implication of the first conjunct, $p_{13} \wedge p_{27}$, is not of type P. The fourth formula clearly is not a Horn formula since it is not a conjunction of implications.

The algorithm we propose for deciding the satisfiability of a Horn formula ϕ maintains a list of all occurrences of type P in ϕ and proceeds like this:

1. It marks $\top$ if it occurs in that list.
2. If there is a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ such that all P_j with $1 \leq j \leq k_i$ are marked, mark P' as well and go to 2. Otherwise (= there is no conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ such that all P_j are marked) go to 3.
3. If $\bot$ is marked, print out 'The Horn formula ϕ is unsatisfiable.' and stop. Otherwise, go to 4.
4. Print out 'The Horn formula ϕ is satisfiable.' and stop.

In these instructions, the markings of formulas are *shared* by all other occurrences of these formulas in the Horn formula. For example, once we mark p_2 because of one of the criteria above, then all other occurrences of p_2 are marked as well. We use pseudo code to specify this algorithm formally:

```
function HORN ($\phi$):
/* precondition: $\phi$ is a Horn formula */
/* postcondition: HORN ($\phi$) decides the satisfiability for $\phi$ */
begin function
    mark all occurrences of $\top$ in $\phi$;
    while there is a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of $\phi$
          such that all $P_j$ are marked but $P'$ isn't do
          mark $P'$
    end while
    if $\bot$ is marked then return 'unsatisfiable' else return 'satisfiable'
end function
```

We need to make sure that this algorithm terminates on all Horn formulas ϕ as input and that its output (= its decision) is always correct.

Theorem 1.47 *The algorithm* HORN *is correct for the satisfiability decision problem of Horn formulas and has no more than $n + 1$ cycles in its while-statement if n is the number of atoms in ϕ. In particular,* HORN *always terminates on correct input.*

PROOF: Let us first consider the question of program termination. Notice that entering the body of the while-statement has the effect of marking an unmarked P which is not $\top$. Since this marking applies to all occurrences of P in ϕ, the while-statement can have at most one more cycle than there are atoms in ϕ.

Since we guaranteed termination, it suffices to show that the answers given by the algorithm HORN are always correct. To that end, it helps to reveal the functional role of those markings. Essentially, marking a P means that that P has got to be true if the formula ϕ is ever going to be satisfiable. We use mathematical induction to show that

$$\text{'All marked } P \text{ are true for all valuations in which } \phi \text{ evaluates to T.'} \quad (1.8)$$

holds after any number of executions of the body of the while-statement above. The base case, zero executions, is when the while-statement has not yet been entered but we already and only marked all occurrences of $\top$. Since $\top$ must be true in all valuations, (1.8) follows.

In the inductive step, we assume that (1.8) holds after k cycles of the while-statement. Then we need to show that same assertion for all marked P after $k + 1$ cycles. If we enter the $(k + 1)$th cycle, the condition of the while-statement is certainly true. Thus, there exists a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ such that all P_j are marked. Let v be any valuation in which ϕ is true. By our induction hypothesis, we know that all P_j and therefore $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i}$ have to be true in v as well. The conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ has be to true in v, too, from which we infer that P' has to be true in v.

By mathematical induction, we therefore secured that (1.8) holds no matter how many cycles that while-statement went through.

Finally, we need to make sure that the if-statement above always renders correct replies. First, if $\perp$ is marked, then there has to be some conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to \perp$ of ϕ such that all P_i are marked as well. By (1.8) that conjunct of ϕ evaluates to $\mathtt{T} \to \mathtt{F} = \mathtt{F}$ whenever ϕ is true. As this is impossible the reply 'unsatisfiable' is correct. Second, if $\perp$ is not marked, we simply assign T to all marked atoms and F to all unmarked atoms and use proof by contradiction to show that ϕ has to be true with respect to that valuation.

If ϕ is *not* true under that valuation, it must make one of its principal conjuncts $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ false. By the semantics of implication this can only mean that all P_j are true and P' is false. By the definition of our valuation, we then infer that all P_j are marked, so $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ is a conjunct of ϕ that would have been dealt with in one of the cycles of the while-statement and so P' is marked, too. Since $\perp$ is not marked, P' has to be $\top$ or some atom q. In any event, the conjunct is then true by (1.8), a contradiction □

Note that the proof by contradiction employed in the last proof was not really needed. It just made the argument seem more natural to us. The literature is full of such examples where one uses proof by contradiction more out of psychological than proof-theoretical necessity.

1.6 SAT solvers

The marking algorithm for Horn formulas computes marks as constraints on all valuations that can make a formule true. By (1.8), all marked atoms have to be true for any such valuation. We can extend this idea to general formulas ϕ by computing constraints saying which subformulas of ϕ require a certain truth value for all valuations that make ϕ true:

'All marked subformulas evaluate to their mark value
for all valuations in which ϕ evaluates to T.' (1.9)

In that way, marking atomic formulas generalizes to marking subformulas; and 'true' marks generalize into 'true' and 'false' marks. At the same time, (1.9) serves as a guide for designing an algorithm and as an invariant for proving its correctness.

1.6.1 A linear solver

We will execute this marking algorithm on the parse tree of formulas, except that we will translate formulas into the adequate fragment

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \tag{1.10}$$

and then share common subformulas of the resulting parse tree, making the tree into a directed, acyclic graph (DAG). The inductively defined translation

$$\begin{array}{ll} T(p) = p & T(\neg\phi) = \neg T(\phi) \\ T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2) & T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2)) \\ T(\phi_1 \rightarrow \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2)) & \end{array}$$

transforms formulas generated by (1.3) into formulas generated by (1.10) such that ϕ and $T(\phi)$ are semantically equivalent and have the same propositional atoms. Therefore, ϕ is satisfiable iff $T(\phi)$ is satisfiable; and the set of valuations for which ϕ is true equals the set of valuations for which $T(\phi)$ is true. The latter ensures that the diagnostics of a SAT solver, applied to $T(\phi)$, is meaningful for the original formula ϕ. In the exercises, you are asked to prove these claims.

Example 1.48 For the formula ϕ being $p \wedge \neg(q \vee \neg p)$ we compute $T(\phi) = p \wedge \neg\neg(\neg q \wedge \neg\neg p)$. The parse tree and DAG of $T(\phi)$ are depicted in Figure 1.12.

Any valuation that makes $p \wedge \neg\neg(\neg q \wedge \neg\neg p)$ true has to assign T to the topmost $\wedge$-node in its DAG of Figure 1.12. But that forces the mark T on the p-node and the topmost $\neg$-node. In the same manner, we arrive at a complete set of constraints in Figure 1.13, where the time stamps '1:' etc indicate the order in which we applied our intuitive reasoning about these constraints; this order is generally not unique.

The formal set of rules for forcing new constraints from old ones is depicted in Figure 1.14. A small circle indicates any node ($\neg$, $\wedge$ or atom). The force laws for negation, $\neg_{\mathrm{t}}$ and $\neg_{\mathrm{f}}$, indicate that a truth constraint on a $\neg$-node forces its dual value at its sub-node and vice versa. The law $\wedge_{\mathrm{te}}$ propagates a T constraint on a $\wedge$-node to its two sub-nodes; dually, $\wedge_{\mathrm{ti}}$ forces a T mark on a $\wedge$-node if both its children have that mark. The laws $\wedge_{\mathrm{fl}}$ and $\wedge_{\mathrm{fr}}$ force a F constraint on a $\wedge$-node if any of its sub-nodes has a F value. The laws $\wedge_{\mathrm{fll}}$

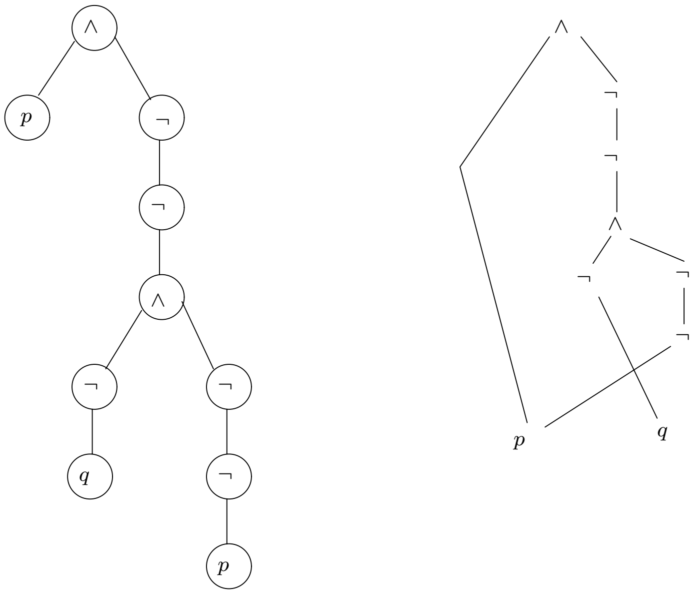


Figure 1.12. Parse tree (left) and directed acyclic graph (right) of the formula from Example 1.48. The p-node is shared on the right.

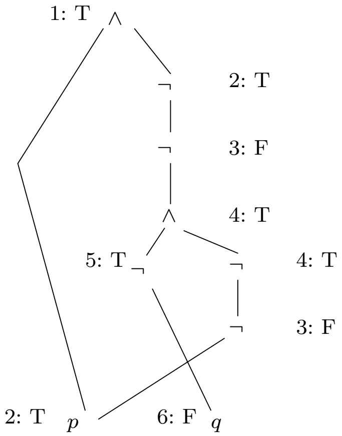
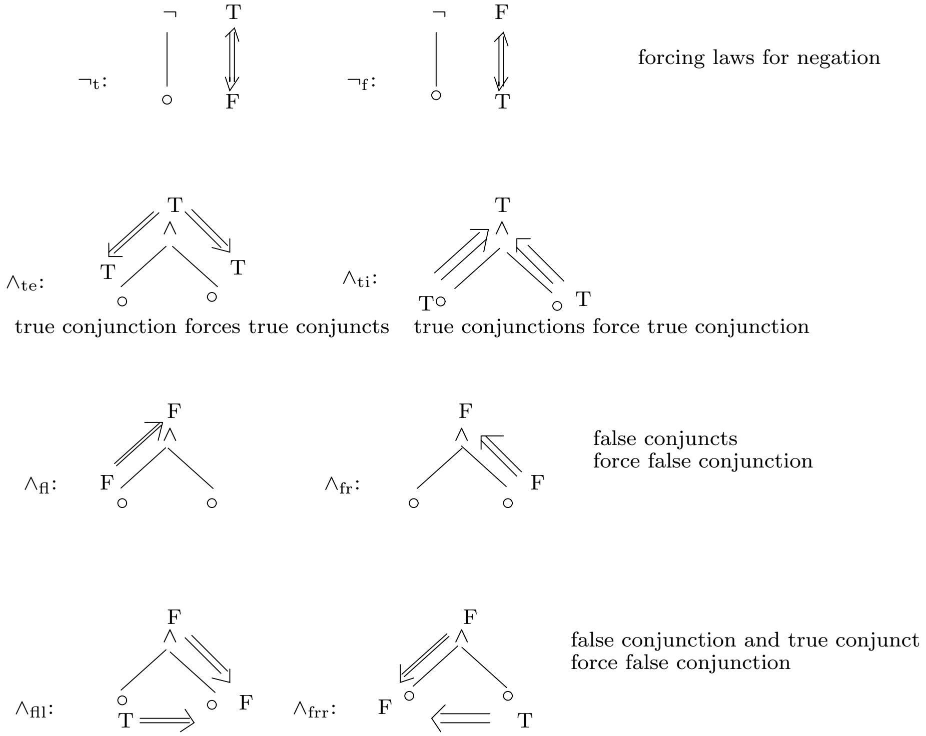


Figure 1.13. A witness to the satisfiability of the formula represented by this DAG.

and $\wedge_{\mathrm{frr}}$ are more complex: if an $\wedge$-node has a F constraint and one of its sub-nodes has a T constraint, then the *other* sub-node obtains a F-constraint. Please check that all constraints depicted in Figure 1.13 are derivable from these rules. The fact that each node in a DAG obtained a forced marking does not yet show that this is a witness to the satisfiability of the formula



forcing laws for negation

true conjunction forces true conjuncts

true conjunctions force true conjunction

false conjuncts force false conjunction

false conjunction and true conjunct force false conjunction

Figure 1.14. Rules for flow of constraints in a formula's DAG. Small circles indicate arbitrary nodes ($\neg$, $\wedge$ or atom). Note that the rules $\wedge_{\mathrm{fll}}$, $\wedge_{\mathrm{frr}}$ and $\wedge_{\mathrm{ti}}$ require that the source constraints of both $\Longrightarrow$ are present.

represented by this DAG. A post-processing phase takes the marks for all atoms and re-computes marks of all other nodes in a bottom-up manner, as done in Section 1.4 on parse trees. Only if the resulting marks match the ones we computed have we found a witness. Please verify that this is the case in Figure 1.13.

We can apply SAT solvers to checking whether sequents are valid. For example, the sequent $p \wedge q \to r \vdash p \to q \to r$ is valid iff $(p \wedge q \to r) \to p \to q \to r$ is a theorem (why?) iff $\phi = \neg((p \wedge q \to r) \to p \to q \to r)$ is *not* satisfiable. The DAG of $T(\phi)$ is depicted in Figure 1.15. The annotations "1" etc indicate which nodes represent which sub-formulas. Notice that such DAGs may be constructed by applying the translation clauses for T to sub-formulas in a bottom-up manner – sharing equal subgraphs were applicable.

The findings of our SAT solver can be seen in Figure 1.16. The solver concludes that the indicated node requires the marks `T` *and* `F` for (1.9) to be met. Such contradictory constraints therefore imply that all formulas $T(\phi)$ whose DAG equals that of this figure are not satisfiable. In particular, all

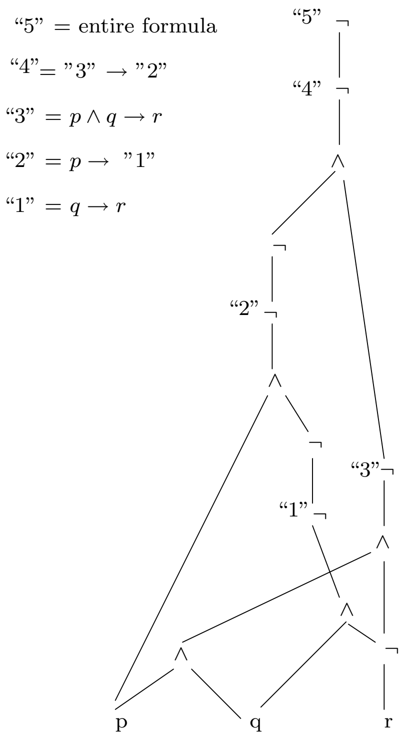


Figure 1.15. The DAG for the translation of $\neg((p \wedge q \rightarrow r) \rightarrow p \rightarrow q \rightarrow r)$. Labels "1" etc indicate which nodes represent what subformulas.

such ϕ are unsatisfiable. This SAT solver has a linear running time in the size of the DAG for $T(\phi)$. Since that size is a linear function of the length of ϕ – the translation T causes only a linear blow-up – our SAT solver has a linear running time in the length of the formula. This linearity came with a price: our linear solver fails for all formulas of the form $\neg(\phi_1 \wedge \phi_2)$.

1.6.2 A cubic solver

When we applied our linear SAT solver, we saw two possible outcomes: we either detected contradictory constraints, meaning that no formula represented by the DAG is satisfiable (e.g. Fig. 1.16); or we managed to force consistent constraints on all nodes, in which case all formulas represented by this DAG are satisfiable with those constraints as a witness (e.g. Fig. 1.13). Unfortunately, there is a third possibility: all forced constraints are consistent with each other, but not all nodes are constrained! We already remarked that this occurs for formulas of the form $\neg(\phi_1 \wedge \phi_2)$.

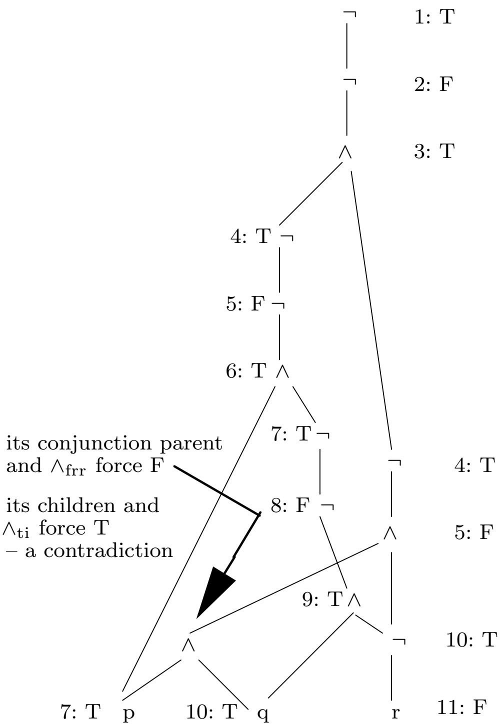


Figure 1.16. The forcing rules, applied to the DAG of Figure 1.15, detect contradictory constraints at the indicated node – implying that the initial constraint '1:T' cannot be realized. Thus, formulas represented by this DAG are not satisfiable.

Recall that checking validity of formulas in CNF is very easy. We already hinted at the fact that checking satisfiability of formulas in CNF is hard. To illustrate, consider the formula

$$((p \vee (q \vee r)) \wedge ((p \vee \neg q) \wedge ((q \vee \neg r) \wedge ((r \vee \neg p) \wedge (\neg p \vee (\neg q \vee \neg r)))))) \tag{1.11}$$

in CNF – based on Example 4.2, page 77, in [Pap94]. Intuitively, this formula should not be satisfiable. The first and last clause in (1.11) 'say' that at least one of p, q, and r are false and true (respectively). The remaining three clauses, in their conjunction, 'say' that p, q, and r all have the same truth value. This cannot be satisfiable, and a good SAT solver should discover this without any user intervention. Unfortunately, our linear SAT solver can neither detect inconsistent constraints nor compute constraints for all nodes. Figure 1.17 depicts the DAG for $T(\phi)$, where ϕ is as in (1.11); and reveals

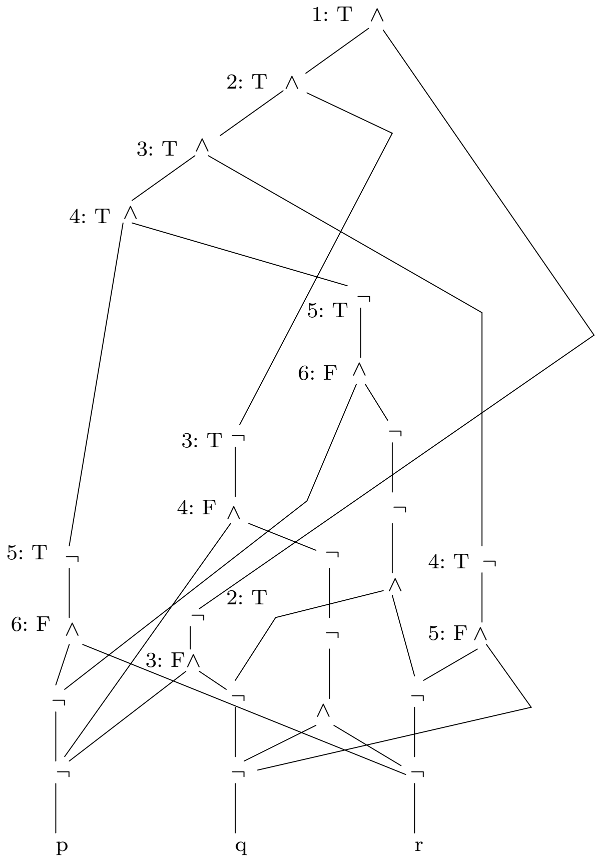
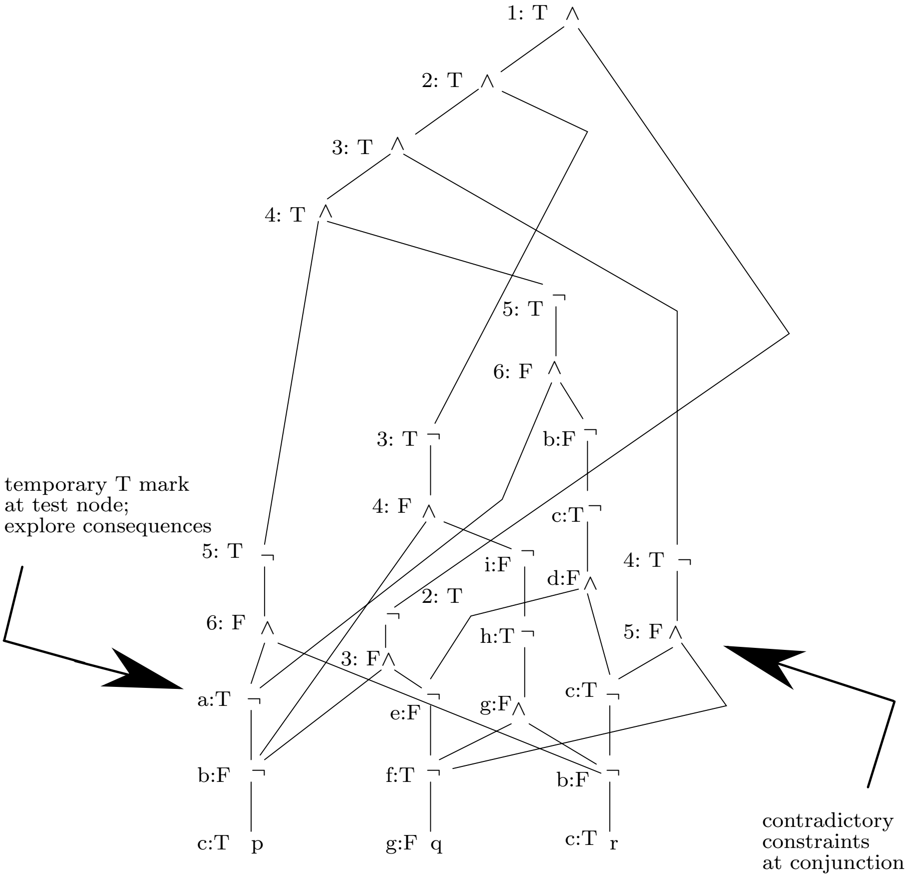


Figure 1.17. The DAG for the translation of the formula in (1.11). It has a $\wedge$-spine of length 4 as it is a conjunction of five clauses. Its linear analysis gets stuck: all forced constraints are consistent with each other but several nodes, including all atoms, are unconstrained.

that our SAT solver got stuck: no inconsistent constraints were found and not all nodes obtained constraints; in particular, no atom received a mark! So how can we improve this analysis? Well, we can mimic the role of LEM to improve the precision of our SAT solver. For the DAG with marks as in Figure 1.17, pick any node n that is not yet marked. Then *test* node n by making two independent computations:

1. determine which *temporary* marks are forced by adding to the marks in Figure 1.17 the `T` mark only to n; and
2. determine which *temporary* marks are forced by adding, again to the marks in Figure 1.17, the `F` mark only to n.



temporary T mark
at test node;
explore consequences

contradictory
constraints
at conjunction

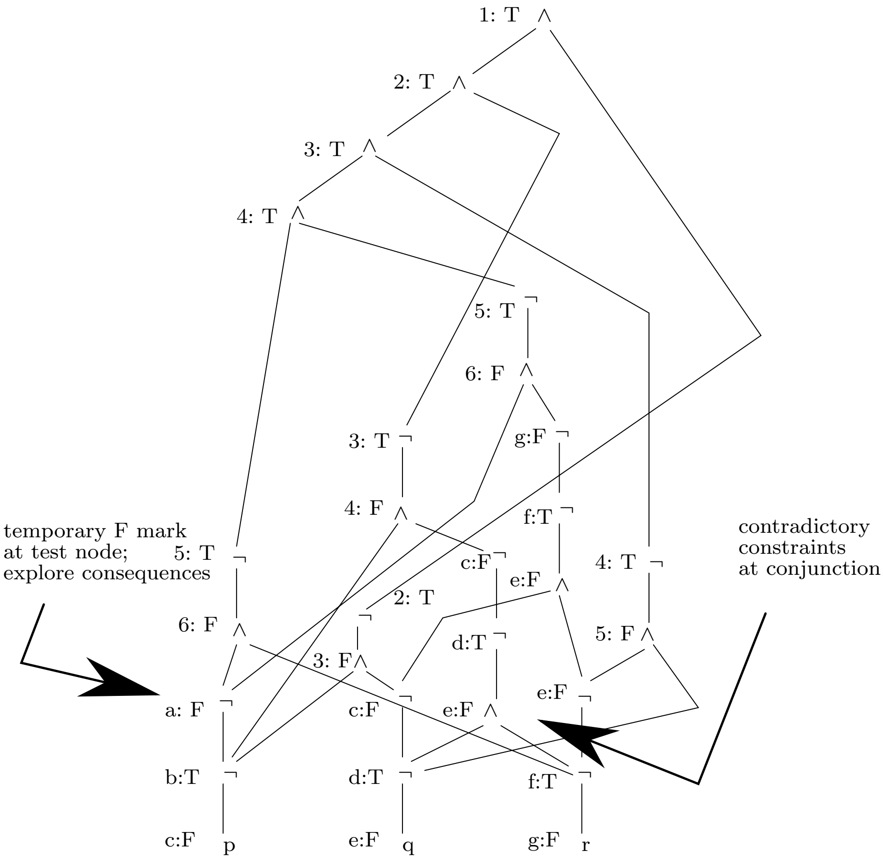
Figure 1.18. Marking an unmarked node with T and exploring what new constraints would follow from this. The analysis shows that this test marking causes contradictory constraints. We use lowercase letters 'a:' etc to denote temporary marks.

If both runs find contradictory constraints, the algorithm stops and reports that $T(\phi)$ is unsatisfiable. Otherwise, all nodes that received the same mark in both of these runs receive that very mark as a *permanent* one; that is, we update the mark state of Figure 1.17 with all such shared marks.

We test any further unmarked nodes in the same manner until we either find contradictory *permanent* marks, a complete witness to satisfiability (all nodes have consistent marks), or we have tested *all* currently unmarked nodes in this manner without detecting any shared marks. Only in the latter case does the analysis terminate without knowing whether the formulas represented by that DAG are satisfiable.

Example 1.49 We revisit our stuck analysis of Figure 1.17. We test a ¬-node and explore the consequences of setting that ¬-node's mark to T; Figure 1.18 shows the result of that analysis. Dually, Figure 1.19 tests the consequences of setting that ¬-node's mark to F. Since both runs reveal a contradiction, the algorithm terminates, ruling that the formula in (1.11) is not satisfiable.

In the exercises, you are asked to show that the specification of our cubic SAT solver is sound. Its running time is indeed cubic in the size of the DAG (and the length of original formula). One factor stems from the linear SAT solver used in each test run. A second factor is introduced since each unmarked node has to be tested. The third factor is needed since each new permanent mark causes *all* unmarked nodes to be tested again.



temporary F mark at test node; explore consequences

contradictory constraints at conjunction

Figure 1.19. Marking the same unmarked node with F and exploring what new constraints would follow from this. The analysis shows that this test marking also causes contradictory constraints.

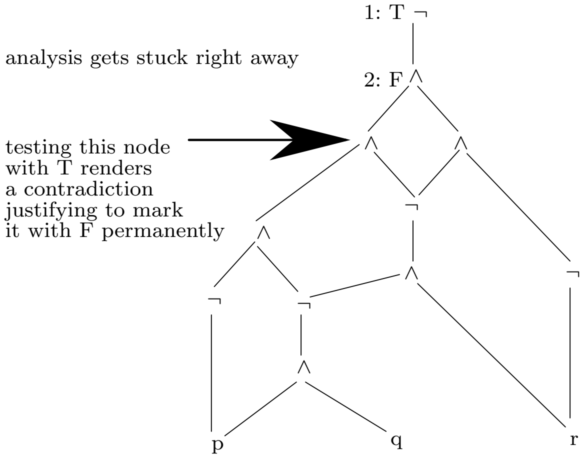


Figure 1.20. Testing the indicated node with `T` causes contradictory constraints, so we may mark that node with `F` permanently. However, our algorithm does not seem to be able to decide satisfiability of this DAG even with that optimization.

We deliberately under-specified our cubic SAT solver, but any implementation or optimization decisions need to secure soundness of the analysis. All replies of the form

1. 'The input formula is not satisfiable' and
2. 'The input formula is satisfiable under the following valuation . . . '

have to be correct. The third form of reply 'Sorry, I could not figure this one out.' is correct by definition. :-) We briefly discuss two sound modifications to the algorithm that introduce some overhead, but may cause the algorithm to decide many more instances. Consider the state of a DAG right after we have explored consequences of a temporary mark on a test node.

1. If that state – permanent plus temporary markings – contains contradictory constraints, we can erase all temporary marks and mark the test node permanently with the dual mark of its test. That is, if marking node n with v resulted in a contradiction, it will get a permanent mark $\overline{v}$, where $\overline{\mathtt{T}} = \mathtt{F}$ and $\overline{\mathtt{F}} = \mathtt{T}$; otherwise
2. if that state managed to mark *all* nodes with consistent constraints, we report these markings as a witness of satisfiability and terminate the algorithm.

If none of these cases apply, we proceed as specified: promote shared marks of the two test runs to permanent ones, if applicable.

Example 1.50 To see how one of these optimizations may make a difference, consider the DAG in Figure 1.20. If we test the indicated node with

T, contradictory constraints arise. Since any witness of satisfiability has to assign some value to that node, we infer that it cannot be T. Thus, we may permanently assign mark F to that node. For this DAG, such an optimization does not seem to help. No test of an unmarked node detects a shared mark or a shared contradiction. Our cubic SAT solver fails for this DAG.

1.7 Exercises

Exercises 1.1

1. Use $\neg$, $\rightarrow$, $\wedge$ and $\vee$ to express the following declarative sentences in propositional logic; in each case state what your respective propositional atoms p, q, etc. mean:
   - * (a) If the sun shines today, then it won't shine tomorrow.
   - (b) Robert was jealous of Yvonne, or he was not in a good mood.
   - (c) If the barometer falls, then either it will rain or it will snow.
   - * (d) If a request occurs, then either it will eventually be acknowledged, or the requesting process won't ever be able to make progress.
   - (e) Cancer will not be cured unless its cause is determined and a new drug for cancer is found.
   - (f) If interest rates go up, share prices go down.
   - (g) If Smith has installed central heating, then he has sold his car or he has not paid his mortgage.
   - * (h) Today it will rain or shine, but not both.
   - * (i) If Dick met Jane yesterday, they had a cup of coffee together, or they took a walk in the park.
   - (j) No shoes, no shirt, no service.
   - (k) My sister wants a black and white cat.
2. The formulas of propositional logic below implicitly assume the binding priorities of the logical connectives put forward in Convention 1.3. Make sure that you fully understand those conventions by reinserting as many brackets as possible. For example, given $p \wedge q \rightarrow r$, change it to $(p \wedge q) \rightarrow r$ since $\wedge$ binds more tightly than $\rightarrow$.
   - * (a) $\neg p \wedge q \rightarrow r$
   - (b) $(p \rightarrow q) \wedge \neg(r \vee p \rightarrow q)$
   - * (c) $(p \rightarrow q) \rightarrow (r \rightarrow s \vee t)$
   - (d) $p \vee (\neg q \rightarrow p \wedge r)$
   - * (e) $p \vee q \rightarrow \neg p \wedge r$
   - (f) $p \vee p \rightarrow \neg q$
   - * (g) Why is the expression $p \vee q \wedge r$ problematic?

Exercises 1.2

1. Prove the validity of the following sequents:
   - (a) $(p \wedge q) \wedge r, s \wedge t \vdash q \wedge s$

(b) $p \wedge q \vdash q \wedge p$

* (c) $(p \wedge q) \wedge r \vdash p \wedge (q \wedge r)$

(d) $p \rightarrow (p \rightarrow q), p \vdash q$

* (e) $q \rightarrow (p \rightarrow r), \neg r, q \vdash \neg p$

* (f) $\vdash (p \wedge q) \rightarrow p$

(g) $p \vdash q \rightarrow (p \wedge q)$

* (h) $p \vdash (p \rightarrow q) \rightarrow q$

* (i) $(p \rightarrow r) \wedge (q \rightarrow r) \vdash p \wedge q \rightarrow r$

* (j) $q \rightarrow r \vdash (p \rightarrow q) \rightarrow (p \rightarrow r)$

(k) $p \rightarrow (q \rightarrow r), p \rightarrow q \vdash p \rightarrow r$

* (l) $p \rightarrow q, r \rightarrow s \vdash p \vee r \rightarrow q \vee s$

(m) $p \vee q \vdash r \rightarrow (p \vee q) \wedge r$

* (n) $(p \vee (q \rightarrow p)) \wedge q \vdash p$

* (o) $p \rightarrow q,\ r \rightarrow s \vdash p \wedge r \rightarrow q \wedge s$

(p) $p \rightarrow q \vdash ((p \wedge q) \rightarrow p) \wedge (p \rightarrow (p \wedge q))$

(q) $\vdash q \rightarrow (p \rightarrow (p \rightarrow (q \rightarrow p)))$

* (r) $p \rightarrow q \wedge r \vdash (p \rightarrow q) \wedge (p \rightarrow r)$

(s) $(p \rightarrow q) \wedge (p \rightarrow r) \vdash p \rightarrow q \wedge r$

(t) $\vdash (p \rightarrow q) \rightarrow ((r \rightarrow s) \rightarrow (p \wedge r \rightarrow q \wedge s))$; here you might be able to 'recycle' and augment a proof from a previous exercise.

(u) $p \rightarrow q \vdash \neg q \rightarrow \neg p$

* (v) $p \vee (p \wedge q) \vdash p$

(w) $r, p \rightarrow (r \rightarrow q) \vdash p \rightarrow (q \wedge r)$

* (x) $p \rightarrow (q \vee r), q \rightarrow s, r \rightarrow s \vdash p \rightarrow s$

* (y) $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$.

2. For the sequents below, show which ones are valid and which ones aren't:

* (a) $\neg p \rightarrow \neg q \vdash q \rightarrow p$

* (b) $\neg p \vee \neg q \vdash \neg(p \wedge q)$

* (c) $\neg p, p \vee q \vdash q$

* (d) $p \vee q, \neg q \vee r \vdash p \vee r$

* (e) $p \rightarrow (q \vee r), \neg q, \neg r \vdash \neg p$ without using the MT rule

* (f) $\neg p \wedge \neg q \vdash \neg(p \vee q)$

* (g) $p \wedge \neg p \vdash \neg(r \rightarrow q) \wedge (r \rightarrow q)$

(h) $p \rightarrow q, s \rightarrow t \vdash p \vee s \rightarrow q \wedge t$

* (i) $\neg(\neg p \vee q) \vdash p$.

3. Prove the validity of the sequents below:

(a) $\neg p \rightarrow p \vdash p$

(b) $\neg p \vdash p \rightarrow q$

(c) $p \vee q, \neg q \vdash p$

* (d) $\vdash \neg p \rightarrow (p \rightarrow (p \rightarrow q))$

(e) $\neg(p \rightarrow q) \vdash q \rightarrow p$

(f) $p \rightarrow q \vdash \neg p \vee q$

(g) $\vdash \neg p \vee q \rightarrow (p \rightarrow q)$

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

(i) $(c \wedge n) \to t,\ h \wedge \neg s,\ h \wedge \neg(s \vee c) \to p \vdash (n \wedge \neg t) \to p$

(j) the two sequents implict in (1.2) on page 20

(k) $q \vdash (p \wedge q) \vee (\neg p \wedge q)$ using LEM

(l) $\neg(p \wedge q) \vdash \neg p \vee \neg q$

(m) $p \wedge q \to r \vdash (p \to r) \vee (q \to r)$

* (n) $p \wedge q \vdash \neg(\neg p \vee \neg q)$

(o) $\neg(\neg p \vee \neg q) \vdash p \wedge q$

(p) $p \to q \vdash \neg p \vee q$ possibly without using LEM?

* (q) $\vdash (p \to q) \vee (q \to r)$ using LEM

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

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

(t) $(p \to q) \to r,\ s \to \neg p,\ t,\ \neg s \wedge t \to q \vdash r$

(u) $(s \to p) \vee (t \to q) \vdash (s \to q) \vee (t \to p)$

(v) $(p \wedge q) \to r,\ r \to s,\ q \wedge \neg s \vdash \neg p.$

4. Explain why intuitionistic logicians also reject the proof rule PBC.

5. Prove the following theorems of propositional logic:

* (a) $((p \to q) \to q) \to ((q \to p) \to p)$

(b) Given a proof for the sequent of the previous item, do you now have a quick argument for $((q \to p) \to p) \to ((p \to q) \to q)$?

(c) $((p \to q) \wedge (q \to p)) \to ((p \vee q) \to (p \wedge q))$

* (d) $(p \to q) \to ((\neg p \to q) \to q)$.

6. Natural deduction is not the only possible formal framework for proofs in propositional logic. As an abbreviation, we write Γ to denote any finite sequence of formulas $\phi_1, \phi_2, \dots, \phi_n$ $(n \geq 0)$. Thus, any sequent may be written as $\Gamma \vdash \psi$ for an appropriate, possibly empty, Γ. In this exercise we propose a different notion of proof, which states rules for transforming valid sequents into valid sequents. For example, if we have already a proof for the sequent $\Gamma, \phi \vdash \psi$, then we obtain a proof of the sequent $\Gamma \vdash \phi \to \psi$ by augmenting this very proof with one application of the rule →i. The new approach expresses this as an inference rule between sequents:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ \to\text{i}.$$

The rule 'assumption' is written as

$$\frac{}{\phi \vdash \phi}\ \text{assumption}$$

i.e. the premise is empty. Such rules are called axioms.

(a) Express all remaining proof rules of Figure 1.2 in such a form. (Hint: some of your rules may have more than one premise.)

(b) Explain why proofs of $\Gamma \vdash \psi$ in this new system have a tree-like structure with $\Gamma \vdash \psi$ as root.

(c) Prove $p \vee (p \wedge q) \vdash p$ in your new proof system.

7. Show that $\sqrt{2}$ cannot be a rational number. Proceed by proof by contradiction: assume that $\sqrt{2}$ is a fraction k/l with integers k and $l \neq 0$. On squaring both sides we get $2 = k^2/l^2$, or equivalently $2l^2 = k^2$. We may assume that any common 2 factors of k and l have been cancelled. Can you now argue that $2l^2$ has a different number of 2 factors from k^2? Why would that be a contradiction and to what?
8. There is an alternative approach to treating negation. One could simply ban the operator $\neg$ from propositional logic and think of $\phi \to \bot$ as 'being' $\neg\phi$. Naturally, such a logic cannot rely on the natural deduction rules for negation. Which of the rules $\neg$i, $\neg$e, $\neg\neg$e and $\neg\neg$i can you simulate with the remaining proof rules by letting $\neg\phi$ be $\phi \to \bot$?
9. Let us introduce a new connective $\phi \leftrightarrow \psi$ which should abbreviate $(\phi \to \psi) \land (\psi \to \phi)$. Design introduction and elimination rules for $\leftrightarrow$ and show that they are derived rules if $\phi \leftrightarrow \psi$ is interpreted as $(\phi \to \psi) \land (\psi \to \phi)$.

Exercises 1.3

In order to facilitate reading these exercises we assume below the usual conventions about binding priorities agreed upon in Convention 1.3.

1. Given the following formulas, draw their corresponding parse tree:
   (a) p
   * (b) $p \land q$
   (c) $p \land \neg q \to \neg p$
   * (d) $p \land (\neg q \to \neg p)$
   (e) $p \to (\neg q \lor (q \to p))$
   * (f) $\neg((\neg q \land (p \to r)) \land (r \to q))$
   (g) $\neg p \lor (p \to q)$
   (h) $(p \land q) \to (\neg r \lor (q \to r))$
   (i) $((s \lor (\neg p)) \to (\neg p))$
   (j) $(s \lor ((\neg p) \to (\neg p)))$
   (k) $(((s \to (r \lor l)) \lor ((\neg q) \land r)) \to ((\neg(p \to s)) \to r))$
   (l) $(p \to q) \land (\neg r \to (q \lor (\neg p \land r)))$.
2. For each formula below, list all its subformulas:
   * (a) $p \to (\neg p \lor (\neg\neg q \to (p \land q)))$
   (b) $(s \to r \lor l) \lor (\neg q \land r) \to (\neg(p \to s) \to r)$
   (c) $(p \to q) \land (\neg r \to (q \lor (\neg p \land r)))$.
3. Draw the parse tree of a formula ϕ of propositional logic which is
   * (a) a negation of an implication
   (b) a disjunction whose disjuncts are both conjunctions
   * (c) a conjunction of conjunctions.
4. For each formula below, draw its parse tree and list all subformulas:
   * (a) $\neg(s \to (\neg(p \to (q \lor \neg s))))$
   (b) $((p \to \neg q) \lor (p \land r) \to s) \lor \neg r$.

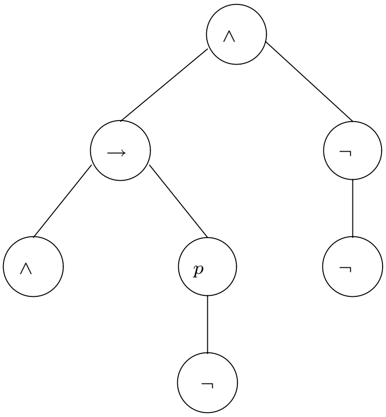


Figure 1.21. A tree that represents an ill-formed formula.

* 5. For the parse tree in Figure 1.22 find the logical formula it represents.

6. For the trees below, find their linear representations and check whether they correspond to well-formed formulas:
   (a) the tree in Figure 1.10 on page 44
   (b) the tree in Figure 1.23.

* 7. Draw a parse tree that represents an ill-formed formula such that
   (a) one can extend it by adding one or several subtrees to obtain a tree that represents a well-formed formula;
   (b) it is inherently ill-formed; i.e. any extension of it could not correspond to a well-formed formula.

8. Determine, by trying to draw parse trees, which of the following formulas are well-formed:
   (a) $p \wedge \neg(p \vee \neg q) \rightarrow (r \rightarrow s)$
   (b) $p \wedge \neg(p \vee q \wedge s) \rightarrow (r \rightarrow s)$
   (c) $p \wedge \neg(p \vee \wedge s) \rightarrow (r \rightarrow s)$.

   Among the ill-formed formulas above which ones, and in how many ways, could you 'fix' by the insertion of brackets only?

———

Exercises 1.4

* 1. Construct the truth table for $\neg p \vee q$ and verify that it coincides with the one for $p \rightarrow q$. (By 'coincide' we mean that the respective columns of T and F values are the same.)

2. Compute the complete truth table of the formula
   * (a) $((p \rightarrow q) \rightarrow p) \rightarrow p$
   (b) represented by the parse tree in Figure 1.3 on page 34

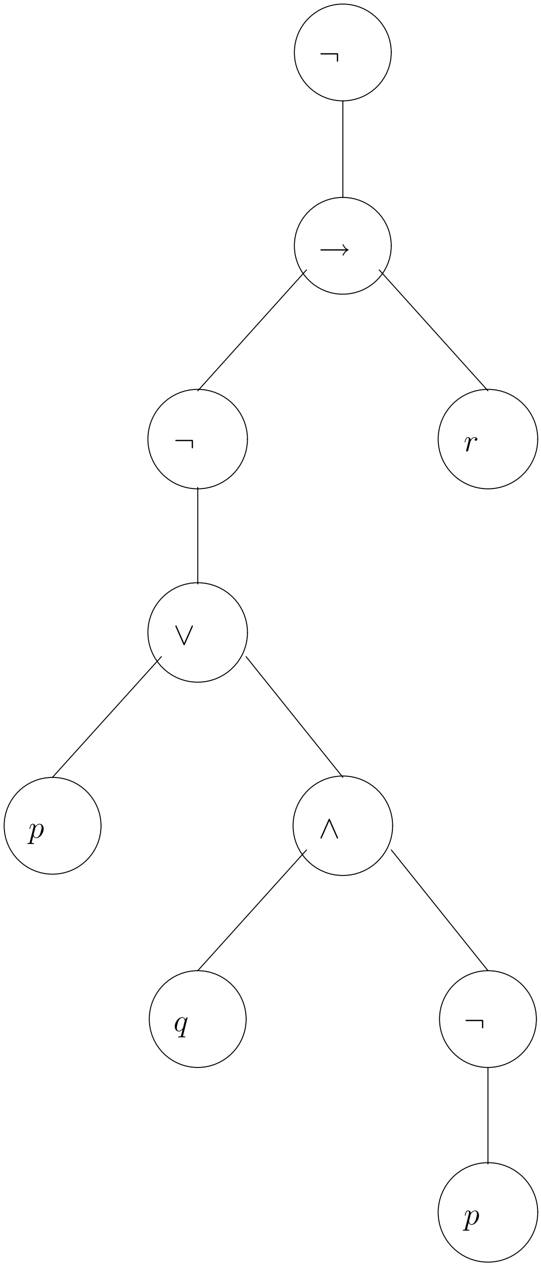


Figure 1.22. A parse tree of a negated implication.

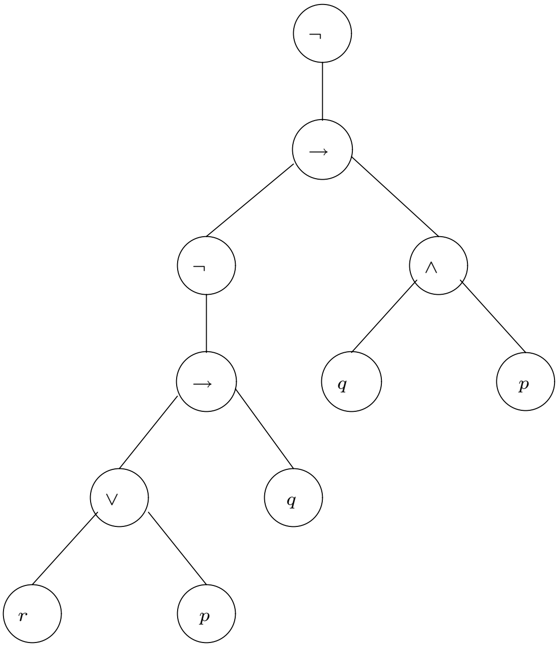


Figure 1.23. Another parse tree of a negated implication.

* (c) $p \vee (\neg(q \wedge (r \rightarrow q)))$
(d) $(p \wedge q) \rightarrow (p \vee q)$
(e) $((p \rightarrow \neg q) \rightarrow \neg p) \rightarrow q$
(f) $(p \rightarrow q) \vee (p \rightarrow \neg q)$
(g) $((p \rightarrow q) \rightarrow p) \rightarrow p$
(h) $((p \vee q) \rightarrow r) \rightarrow ((p \rightarrow r) \vee (q \rightarrow r))$
(i) $(p \rightarrow q) \rightarrow (\neg p \rightarrow \neg q)$.

3. Given a valuation and a parsetree of a formula, compute the truth value of the formula for that valuation (as done in a bottom-up fashion in Figure 1.7 on page 40) with the parse tree in

* (a) Figure 1.10 on page 44 and the valuation in which q and r evaluate to T and p to F;
(b) Figure 1.4 on page 36 and the valuation in which q evaluates to T and p and r evaluate to F;
(c) Figure 1.23 where we let p be T, q be F and r be T; and
(d) Figure 1.23 where we let p be F, q be T and r be F.

4. Compute the truth value on the formula's parse tree, or specify the corresponding line of a truth table where

* (a) p evaluates to F, q to T and the formula is $p \rightarrow (\neg q \vee (q \rightarrow p))$
* (b) the formula is $\neg((\neg q \wedge (p \rightarrow r)) \wedge (r \rightarrow q))$, p evaluates to F, q to T and r evaluates to T.

* 5. A formula is valid iff it computes T for all its valuations; it is satisfiable iff it computes T for at least one of its valuations. Is the formula of the parse tree in Figure 1.10 on page 44 valid? Is it satisfiable?

6. Let $*$ be a new logical connective such that $p * q$ does not hold iff p and q are either both false or both true.
   (a) Write down the truth table for $p * q$.
   (b) Write down the truth table for $(p * p) * (q * q)$.
   (c) Does the table in (b) coincide with a table in Figure 1.6 (page 38)? If so, which one?
   (d) Do you know $*$ already as a logic gate in circuit design? If so, what is it called?

7. These exercises let you practice proofs using mathematical induction. Make sure that you state your base case and inductive step clearly. You should also indicate where you apply the induction hypothesis.
   (a) Prove that

$$(2 \cdot 1 - 1) + (2 \cdot 2 - 1) + (2 \cdot 3 - 1) + \cdots + (2 \cdot n - 1) = n^2$$

   by mathematical induction on $n \geq 1$.
   (b) Let k and l be natural numbers. We say that k is divisible by l if there exists a natural number p such that $k = p \cdot l$. For example, 15 is divisible by 3 because $15 = 5 \cdot 3$. Use mathematical induction to show that $11^n - 4^n$ is divisible by 7 for all natural numbers $n \geq 1$.
   * (c) Use mathematical induction to show that

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

   for all natural numbers $n \geq 1$.
   * (d) Prove that $2^n \geq n + 12$ for all natural numbers $n \geq 4$. Here the base case is $n = 4$. Is the statement true for any $n < 4$?
   (e) Suppose a post office sells only 2¢ and 3¢ stamps. Show that any postage of 2¢, or over, can be paid for using only these stamps. Hint: use mathematical induction on n, where n¢ is the postage. In the inductive step consider two possibilities: first, n¢ can be paid for using only 2¢ stamps. Second, paying n¢ requires the use of at least one 3¢ stamp.
   (f) Prove that for every prefix of a well-formed propositional logic formula the number of left brackets is greater or equal to the number of right brackets.

* 8. The Fibonacci numbers are most useful in modelling the growth of populations. We define them by $F_1 \stackrel{\text{def}}{=} 1$, $F_2 \stackrel{\text{def}}{=} 1$ and $F_{n+1} \stackrel{\text{def}}{=} F_n + F_{n-1}$ for all $n \geq 2$. So $F_3 \stackrel{\text{def}}{=} F_1 + F_2 = 1 + 1 = 2$ etc. Show the assertion 'F_{3n} is even.' by mathematical induction on $n \geq 1$. Note that this assertion is saying that the sequence $F_3, F_6, F_9, \ldots$ consists of even numbers only.

9. Consider the function rank, defined by

$$\begin{aligned} \mathsf{rank}(p) &\stackrel{\text{def}}{=} 1 \\ \mathsf{rank}(\neg\phi) &\stackrel{\text{def}}{=} 1 + \mathsf{rank}(\phi) \\ \mathsf{rank}(\phi \circ \psi) &\stackrel{\text{def}}{=} 1 + \max(\mathsf{rank}(\phi), \mathsf{rank}(\psi)) \end{aligned}$$

where p is any atom, $\circ \in \{\rightarrow, \vee, \wedge\}$ and $\max(n, m)$ is n if $n \geq m$ and m otherwise. Recall the concept of the height of a formula (Definition 1.32 on page 44). Use mathematical induction on the height of ϕ to show that $\mathsf{rank}(\phi)$ is nothing but the height of ϕ for all formulas ϕ of propositional logic.

* 10. Here is an example of why we need to secure the base case for mathematical induction. Consider the assertion

'The number $n^2 + 5n + 1$ is even for all $n \geq 1$.'

(a) Prove the inductive step of that assertion.
(b) Show that the base case fails to hold.
(c) Conclude that the assertion is false.
(d) Use mathematical induction to show that $n^2 + 5n + 1$ is odd for all $n \geq 1$.

11. For the soundness proof of Theorem 1.35 on page 46,
(a) explain why we could not use mathematical induction but had to resort to course-of-values induction;
(b) give justifications for all inferences that were annotated with 'why?' and
(c) complete the case analysis ranging over the final proof rule applied; inspect the summary of natural deduction rules in Figure 1.2 on page 27 to see which cases are still missing. Do you need to include derived rules?

12. Show that the following sequents are not valid by finding a valuation in which the truth values of the formulas to the left of $\vdash$ are $\mathtt{T}$ and the truth value of the formula to the right of $\vdash$ is $\mathtt{F}$.
(a) $\neg p \vee (q \rightarrow p) \vdash \neg p \wedge q$
(b) $\neg r \rightarrow (p \vee q), r \wedge \neg q \vdash r \rightarrow q$
* (c) $p \rightarrow (q \rightarrow r) \vdash p \rightarrow (r \rightarrow q)$
(d) $\neg p, p \vee q \vdash \neg q$
(e) $p \rightarrow (\neg q \vee r), \neg r \vdash \neg q \rightarrow \neg p$.

13. For each of the following invalid sequents, give examples of natural language declarative sentences for the atoms p, q and r such that the premises are true, but the conclusion false.
* (a) $p \vee q \vdash p \wedge q$
* (b) $\neg p \rightarrow \neg q \vdash \neg q \rightarrow \neg p$
(c) $p \rightarrow q \vdash p \vee q$
(d) $p \rightarrow (q \vee r) \vdash (p \rightarrow q) \wedge (p \rightarrow r)$.

14. Find a formula of propositional logic ϕ which contains only the atoms p, q and r and which is true only when p and q are false, or when $\neg q \wedge (p \vee r)$ is true.

15. Use mathematical induction on n to prove the theorem $((\phi_1 \land (\phi_2 \land (\cdots \land \phi_n)\ldots) \to \psi) \to (\phi_1 \to (\phi_2 \to (\ldots(\phi_n \to \psi)\ldots))))$.
16. Prove the validity of the following sequents needed to secure the completeness result for propositional logic:
    (a) $\phi_1 \land \neg\phi_2 \vdash \neg(\phi_1 \to \phi_2)$
    (b) $\neg\phi_1 \land \neg\phi_2 \vdash \phi_1 \to \phi_2$
    (c) $\neg\phi_1 \land \phi_2 \vdash \phi_1 \to \phi_2$
    (d) $\phi_1 \land \phi_2 \vdash \phi_1 \to \phi_2$
    (e) $\neg\phi_1 \land \phi_2 \vdash \neg(\phi_1 \land \phi_2)$
    (f) $\neg\phi_1 \land \neg\phi_2 \vdash \neg(\phi_1 \land \phi_2)$
    (g) $\phi_1 \land \neg\phi_2 \vdash \neg(\phi_1 \land \phi_2)$
    (h) $\neg\phi_1 \land \neg\phi_2 \vdash \neg(\phi_1 \lor \phi_2)$
    (i) $\phi_1 \land \phi_2 \vdash \phi_1 \lor \phi_2$
    (j) $\neg\phi_1 \land \phi_2 \vdash \phi_1 \lor \phi_2$
    (k) $\phi_1 \land \neg\phi_2 \vdash \phi_1 \lor \phi_2$.
17. Does $\vDash \phi$ hold for the ϕ below? Please justify your answer.
    (a) $(p \to q) \lor (q \to r)$
    * (b) $((q \to (p \lor (q \to p))) \lor \neg(p \to q)) \to p$.

Exercises 1.5

1. Show that a formula ϕ is valid iff $\top \equiv \phi$, where $\top$ is an abbreviation for an instance $p \lor \neg p$ of LEM.
2. Which of these formulas are semantically equivalent to $p \to (q \lor r)$?
    (a) $q \lor (\neg p \lor r)$
    * (b) $q \land \neg r \to p$
    (c) $p \land \neg r \to q$
    * (d) $\neg q \land \neg r \to \neg p$.
3. An adequate set of connectives for propositional logic is a set such that for every formula of propositional logic there is an equivalent formula with only connectives from that set. For example, the set $\{\neg, \lor\}$ is adequate for propositional logic, because any occurrence of $\land$ and $\to$ can be removed by using the equivalences $\phi \to \psi \equiv \neg\phi \lor \psi$ and $\phi \land \psi \equiv \neg(\neg\phi \lor \neg\psi)$.
    (a) Show that $\{\neg, \land\}$, $\{\neg, \to\}$ and $\{\to, \bot\}$ are adequate sets of connectives for propositional logic. (In the latter case, we are treating $\bot$ as a nullary connective.)
    (b) Show that, if $C \subseteq \{\neg, \land, \lor, \to, \bot\}$ is adequate for propositional logic, then $\neg \in C$ or $\bot \in C$. (Hint: suppose C contains neither $\neg$ nor $\bot$ and consider the truth value of a formula ϕ, formed by using only the connectives in C, for a valuation in which every atom is assigned T.)
    (c) Is $\{\leftrightarrow, \neg\}$ adequate? Prove your answer.
4. Use soundness or completeness to show that a sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ has a proof iff $\phi_1 \to \phi_2 \to \ldots \phi_n \to \psi$ is a tautology.

5. Show that the relation $\equiv$ is
   (a) reflexive: $\phi \equiv \phi$ holds for all ϕ
   (b) symmetric: $\phi \equiv \psi$ implies $\psi \equiv \phi$ and
   (c) transitive: $\phi \equiv \psi$ and $\psi \equiv \eta$ imply $\phi \equiv \eta$.
6. Show that, with respect to $\equiv$,
   (a) $\wedge$ and $\vee$ are idempotent:
      i. $\phi \wedge \phi \equiv \phi$
      ii. $\phi \vee \phi \equiv \phi$
   (b) $\wedge$ and $\vee$ are commutative:
      i. $\phi \wedge \psi \equiv \psi \wedge \phi$
      ii. $\phi \vee \psi \equiv \psi \vee \phi$
   (c) $\wedge$ and $\vee$ are associative:
      i. $\phi \wedge (\psi \wedge \eta) \equiv (\phi \wedge \psi) \wedge \eta$
      ii. $\phi \vee (\psi \vee \eta) \equiv (\phi \vee \psi) \vee \eta$
   (d) $\wedge$ and $\vee$ are absorptive:
   * i. $\phi \wedge (\phi \vee \eta) \equiv \phi$
      ii. $\phi \vee (\phi \wedge \eta) \equiv \phi$
   (e) $\wedge$ and $\vee$ are distributive:
      i. $\phi \wedge (\psi \vee \eta) \equiv (\phi \wedge \psi) \vee (\phi \wedge \eta)$
   * ii. $\phi \vee (\psi \wedge \eta) \equiv (\phi \vee \psi) \wedge (\phi \vee \eta)$
   (f) $\equiv$ allows for double negation: $\phi \equiv \neg\neg\phi$ and
   (g) $\wedge$ and $\vee$ satisfies the de Morgan rules:
      i. $\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$
   * ii. $\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$.
7. Construct a formula in CNF based on each of the following truth tables:
   * (a)

p	q	ϕ_1
T	T	F
F	T	F
T	F	F
F	F	T

   * (b)

p	q	r	ϕ_2
T	T	T	T
T	T	F	F
T	F	T	F
F	T	T	T
T	F	F	F
F	T	F	F
F	F	T	T
F	F	F	F

(c)

r	s	q	ϕ_3
T	T	T	F
T	T	F	T
T	F	T	F
F	T	T	F
T	F	F	T
F	T	F	F
F	F	T	F
F	F	F	T

* 8. Write a recursive function IMPL_FREE which requires a (parse tree of a) propositional formula as input and produces an equivalent implication-free formula as output. How many clauses does your case statement need? Recall Definition 1.27 on page 32.

* 9. Compute CNF (NNF (IMPL_FREE $\neg(p \to (\neg(q \land (\neg p \to q))))))$.

10. Use structural induction on the grammar of formulas in CNF to show that the 'otherwise' case in calls to DISTR applies iff both η_1 and η_2 are of type D in (1.6) on page 55.

11. Use mathematical induction on the height of ϕ to show that the call CNF (NNF (IMPL_FREE ϕ)) returns, up to associativity, ϕ if the latter is already in CNF.

12. Why do the functions CNF and DISTR preserve NNF and why is this important?

13. For the call CNF (NNF (IMPL_FREE (ϕ))) on a formula ϕ of propositional logic, explain why
   (a) its output is always a formula in CNF
   (b) its output is semantically equivalent to ϕ
   (c) that call always terminates.

14. Show that all the algorithms presented in Section 1.5.2 terminate on any input meeting their precondition. Can you formalise some of your arguments? Note that algorithms might not call themselves again on formulas with smaller height. E.g. the call of CNF ($\phi_1 \lor \phi_2$) results in a call DISTR (CNF(ϕ_1), CNF(ϕ_2)), where CNF(ϕ_i) may have greater height than ϕ_i. Why is this not a problem?

15. Apply algorithm HORN from page 66 to each of these Horn formulas:
   * (a) $(p \land q \land w \to \bot) \land (t \to \bot) \land (r \to p) \land (\top \to r) \land (\top \to q) \land (u \to s) \land (\top \to u)$
   (b) $(p \land q \land w \to \bot) \land (t \to \bot) \land (r \to p) \land (\top \to r) \land (\top \to q) \land (r \land u \to w) \land (u \to s) \land (\top \to u)$
   (c) $(p \land q \land s \to p) \land (q \land r \to p) \land (p \land s \to s)$
   (d) $(p \land q \land s \to \bot) \land (q \land r \to p) \land (\top \to s)$
   (e) $(p_5 \to p_{11}) \land (p_2 \land p_3 \land p_5 \to p_{13}) \land (\top \to p_5) \land (p_5 \land p_{11} \to \bot)$
   (f) $(\top \to q) \land (\top \to s) \land (w \to \bot) \land (p \land q \land s \to \bot) \land (v \to s) \land (\top \to r) \land (r \to p)$

* (g) $(\top \to q) \land (\top \to s) \land (w \to \bot) \land (p \land q \land s \to v) \land (v \to s) \land (\top \to r) \land (r \to p)$.

16. Explain why the algorithm `HORN` fails to work correctly if we change the concept of Horn formulas by extending the clause for P on page 65 to $P ::= \bot \mid \top \mid p \mid \neg p$?
17. What can you say about the CNF of Horn formulas. More precisely, can you specify syntactic criteria for a CNF that ensure that there is an equivalent Horn formula? Can you describe informally programs which would translate from one form of representation into another?

Exercises 1.6

1. Use mathematical induction to show that, for all ϕ of (1.3) on page 33,
   (a) $T(\phi)$ can be generated by (1.10) on page 69,
   (b) $T(\phi)$ has the same set of valuations as ϕ, and
   (c) the set of valuations in which ϕ is true equals the set of valuations in which $T(\phi)$ is true.

* 2. Show that all rules of Figure 1.14 (page 71) are sound: if all current marks satisfy the invariant (1.9) from page 68, then this invariant still holds if the derived constraint of that rule becomes an additional mark.

3. In Figure 1.16 on page 73 we detected a contradiction which secured the validity of the sequent $p \land q \to r \vdash p \to q \to r$. Use the same method with the linear SAT solver to show that the sequent $\vdash (p \to q) \lor (r \to p)$ is valid. (This is interesting since we proved this validity in natural deduction with a judicious choice of the proof rule LEM; and the linear SAT solver does not employ any case analysis.)

* 4. Consider the sequent $p \lor q, p \to r \vdash r$. Determine a DAG which is not satisfiable iff this sequent is valid. Tag the DAG's root node with '1: T,' apply the forcing laws to it, and extract a witness to the DAG's satisfiability. Explain in what sense this witness serves as an explanation for the fact that $p \lor q, p \to r \vdash r$ is not valid.

5. Explain in what sense the SAT solving technique, as presented in this chapter, can be used to check whether formulas are tautologies.
6. For ϕ from (1.10), can one reverse engineer ϕ from the DAG of $T(\phi)$?
7. Consider a modification of our method which initially tags a DAG's root node with '1: F.' In that case,
   (a) are the forcing laws still sound? If so, state the invariant.
   (b) what can we say about the formula(s) a DAG represents if
      i. we detect contradictory constraints?
      ii. we compute consistent forced constraints for each node?
8. Given an arbitrary Horn formula ϕ, compare our linear SAT solver – applied to $T(\phi)$ – to the marking algorithm – applied to ϕ. Discuss similarities and differences of these approaches.

9. Consider Figure 1.20 on page 77. Verify that
   (a) its test produces contradictory constraints
   (b) its cubic analysis does not decide satisfiability, regardless of whether the two optimizations we described are present.
10. Verify that the DAG of Figure 1.17 (page 74) is indeed the one obtained for $T(\phi)$, where ϕ is the formula in (1.11) on page 73.
* 11. An implementor may be concerned with the possibility that the answers to the cubic SAT solver may depend on a particular order in which we test unmarked nodes or use the rules in Figure 1.14. Give a semi-formal argument for why the analysis results don't depend on such an order.
12. Find a formula ϕ such that our cubic SAT solver cannot decide the satisfiability of $T(\phi)$.
13. **Advanced Project:** Write a complete implementation of the cubic SAT solver described in Section 1.6.2. It should read formulas from the keyboard or a file; should assume right-associativity of $\vee$, $\wedge$, and $\rightarrow$ (respectively); compute the DAG of $T(\phi)$; perform the cubic SAT solver next. Think also about including appropriate user output, diagnostics, and optimizations.
14. Show that our cubic SAT solver specified in this section
   (a) terminates on all syntactically correct input;
   (b) satisfies the invariant (1.9) after the first permanent marking;
   (c) preserves (1.9) for all permanent markings it makes;
   (d) computes only correct satisfiability witnesses;
   (e) computes only correct 'not satisfiable' replies; and
   (f) remains to be correct under the two modifications described on page 77 for handling results of a node's two test runs.

1.8 Bibliographic notes

Logic has a long history stretching back at least 2000 years, but the truth-value semantics of propositional logic presented in this and every logic textbook today was invented only about 160 years ago, by G. Boole [Boo54]. Boole used the symbols + and · for disjunction and conjunction.

Natural deduction was invented by G. Gentzen [Gen69], and further developed by D. Prawitz [Pra65]. Other proof systems existed before then, notably axiomatic systems which present a small number of axioms together with the rule *modus ponens* (which we call $\rightarrow$e). Proof systems often present as small a number of axioms as possible; and only for an adequate set of connectives such as $\rightarrow$ and $\neg$. This makes them hard to use in practice. Gentzen improved the situation by inventing the idea of working with assumptions (used by the rules $\rightarrow$i, $\neg$i and $\vee$e) and by treating all the connectives separately.

Our linear and cubic SAT solvers are variants of Stålmarck's method [SS90], a SAT solver which is patented in Sweden and in the United States of America.

Further historical remarks, and also pointers to other contemporary books about propositional and predicate logic, can be found in the bibliographic remarks at the end of Chapter 2. For an introduction to algorithms and data structures see e.g. [Wei98].

2
Predicate logic

2.1 The need for a richer language

In the first chapter, we developed propositional logic by examining it from three different angles: its proof theory (the natural deduction calculus), its syntax (the tree-like nature of formulas) and its semantics (what these formulas actually mean). From the outset, this enterprise was guided by the study of declarative sentences, statements about the world which can, for every valuation or model, be given a truth value.

We begin this second chapter by pointing out the limitations of propositional logic with respect to encoding declarative sentences. Propositional logic dealt quite satisfactorily with sentence components like *not*, *and*, *or* and *if* ... *then*, but the logical aspects of natural and artificial languages are much richer than that. What can we do with modifiers like *there exists* ..., *all* ..., *among* ... and *only* ...? Here, propositional logic shows clear limitations and the desire to express more subtle declarative sentences led to the design of *predicate logic*, which is also called *first-order logic*.

Let us consider the declarative sentence

$$\textit{Every student is younger than some instructor.} \tag{2.1}$$

In propositional logic, we could identify this assertion with a propositional atom p. However, that fails to reflect the finer logical structure of this sentence. What is this statement about? Well, it is about *being a student*, *being an instructor* and *being younger than somebody else*. These are all properties of some sort, so we would like to have a mechanism for expressing them together with their logical relationships and dependences.

We now use *predicates* for that purpose. For example, we could write $S(andy)$ to denote that Andy is a student and $I(paul)$ to say that Paul is an instructor. Likewise, $Y(andy, paul)$ could mean that Andy is younger than

Paul. The symbols S, I and Y are called predicates. Of course, we have to be clear about their meaning. The predicate Y could have meant that the second person is younger than the first one, so we need to specify exactly what these symbols refer to.

Having such predicates at our disposal, we still need to formalise those parts of the sentence above which speak of *every* and *some*. Obviously, this sentence refers to the individuals that make up some academic community (left implicit by the sentence), like Kansas State University or the University of Birmingham, and it says that for each student among them there is an instructor among them such that the student is younger than the instructor.

These predicates are not yet enough to allow us to express the sentence in (2.1). We don't really want to write down all instances of $S(\cdot)$ where $\cdot$ is replaced by every student's name in turn. Similarly, when trying to codify a sentence having to do with the execution of a program, it would be rather laborious to have to write down every state of the computer. Therefore, we employ the concept of a *variable*. Variables are written $u, v, w, x, y, z, \dots$ or $x_1, y_3, u_5, \dots$ and can be thought of as *place holders* for concrete values (like a student, or a program state). Using variables, we can now specify the meanings of S, I and Y more formally:

$$\begin{aligned} S(x):&\quad x \text{ is a student}\\ I(x):&\quad x \text{ is an instructor}\\ Y(x,y):&\quad x \text{ is younger than } y. \end{aligned}$$

Note that the names of the variables are not important, provided that we use them consistently. We can state the intended meaning of I by writing

$$I(y): \quad y \text{ is an instructor}$$

or, equivalently, by writing

$$I(z): \quad z \text{ is an instructor.}$$

Variables are mere place holders for objects. The availability of variables is still not sufficient for capturing the essence of the example sentence above. We need to convey the meaning of '**Every** *student x is younger than* **some** *instructor y.'* This is where we need to introduce *quantifiers* $\forall$ (read: 'for all') and $\exists$ (read: 'there exists' or 'for some') which always come attached to a variable, as in $\forall x$ ('for all x') or in $\exists z$ ('there exists z', or 'there is some z'). Now we can write the example sentence in an entirely symbolic way as

$$\forall x\,(S(x) \to (\exists y\,(I(y) \wedge Y(x,y)))).$$

Actually, this encoding is rather a paraphrase of the original sentence. In our example, the re-translation results in

> *For every x, if x is a student, then there is some y which is an instructor such that x is younger than y.*

Different predicates can have a different number of arguments. The predicates S and I have just one (they are called *unary predicates*), but predicate Y requires two arguments (it is called a *binary predicate*). Predicates with any finite number of arguments are possible in predicate logic.

Another example is the sentence

Not all birds can fly.

For that we choose the predicates B and F which have one argument expressing

$$\begin{aligned} B(x): &\quad x \text{ is a bird} \\ F(x): &\quad x \text{ can fly.} \end{aligned}$$

The sentence 'Not all birds can fly' can now be coded as

$$\neg(\forall x\,(B(x) \to F(x)))$$

saying: 'It is not the case that all things which are birds can fly.' Alternatively, we could code this as

$$\exists x\,(B(x) \land \neg F(x))$$

meaning: 'There is some x which is a bird and cannot fly.' Note that the first version is closer to the linguistic structure of the sentence above. These two formulas should evaluate to T in the world we currently live in since, for example, penguins are birds which cannot fly. Shortly, we address how such formulas can be given their meaning in general. We will also explain why formulas like the two above are indeed equivalent *semantically*.

Coding up complex facts expressed in English sentences as logical formulas in predicate logic is important – e.g. in software design with UML or in formal specification of safety-critical systems – and much more care must be taken than in the case of propositional logic. However, once this translation has been accomplished our main objective is to reason symbolically ($\vdash$) or semantically ($\vDash$) about the information expressed in those formulas.

In Section 2.3, we extend our natural deduction calculus of propositional logic so that it covers logical formulas of predicate logic as well. In this way we are able to prove the validity of sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ in a similar way to that in the first chapter.

In Section 2.4, we generalize the valuations of Chapter 1 to a proper notion of models, real or artificial worlds in which formulas of predicate logic can be true or false, which allows us to define semantic entailment $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$.

The latter expresses that, given *any* such model in which all $\phi_1, \phi_2, \dots, \phi_n$ hold, it is the case that ψ holds in that model as well. In that case, one also says that ψ is *semantically entailed* by $\phi_1, \phi_2, \dots, \phi_n$. Although this definition of semantic entailment closely matches the one for propositional logic in Definition 1.34, the process of *evaluating a predicate formula* differs from the computation of truth values for propositional logic in the treatment of predicates (and functions). We discuss it in detail in Section 2.4.

It is outside the scope of this book to show that the natural deduction calculus for predicate logic is sound and complete with respect to semantic entailment; but it is indeed the case that

$$\phi_1, \phi_2, \dots, \phi_n \vdash \psi \quad \text{iff} \quad \phi_1, \phi_2, \dots, \phi_n \vDash \psi$$

for formulas of the predicate calculus. The first proof of this was done by the mathematician K. Gödel.

What kind of reasoning must predicate logic be able to support? To get a feel for that, let us consider the following argument:

> *No books are gaseous. Dictionaries are books. Therefore, no dictionary is gaseous.*

The predicates we choose are

$$\begin{aligned} B(x): &\quad x \text{ is a book} \\ G(x): &\quad x \text{ is gaseous} \\ D(x): &\quad x \text{ is a dictionary.} \end{aligned}$$

Evidently, we need to build a proof theory and semantics that allow us to derive the validity and semantic entailment, respectively, of

$$\neg\exists x\,(B(x) \wedge G(x)),\ \forall x\,(D(x) \to B(x)) \vdash \neg\exists x\,(D(x) \wedge G(x))$$
$$\neg\exists x\,(B(x) \wedge G(x)),\ \forall x\,(D(x) \to B(x)) \vDash \neg\exists x\,(D(x) \wedge G(x)).$$

Verify that these sequents express the argument above in a symbolic form. Predicate logic extends propositional logic not only with quantifiers but with one more concept, that of *function symbols*. Consider the declarative sentence

> *Every child is younger than its mother.*

Using predicates, we could express this sentence as

$$\forall x\, \forall y\, (C(x) \land M(y,x) \to Y(x,y))$$

where $C(x)$ means that x is a child, $M(x,y)$ means that x is y's mother and $Y(x,y)$ means that x is younger than y. (Note that we actually used $M(y,x)$ (y is x's mother), not $M(x,y)$.) As we have coded it, the sentence says that, for all children x and any mother y of theirs, x is younger than y. It is not very elegant to say 'any of x's mothers', since we know that every individual has one and only one mother[1]. The inelegance of coding 'mother' as a predicate is even more apparent if we consider the sentence

Andy and Paul have the same maternal grandmother.

which, using 'variables' a and p for Andy and Paul and a binary predicate M for mother as before, becomes

$$\forall x\, \forall y\, \forall u\, \forall v\, (M(x,y) \land M(y,a) \land M(u,v) \land M(v,p) \to x = u).$$

This formula says that, if y and v are Andy's and Paul's mothers, respectively, and x and u are *their* mothers (i.e. Andy's and Paul's maternal grandmothers, respectively), then x and u are the same person. Notice that we used a special predicate in predicate logic, *equality*; it is a binary predicate, i.e. it takes two arguments, and is written $=$. Unlike other predicates, it is usually written in between its arguments rather than before them; that is, we write $x = y$ instead of $= (x, y)$ to say that x and y are equal.

The function symbols of predicate logic give us a way of avoiding this ugly encoding, for they allow us to represent y's mother in a more direct way. Instead of writing $M(x,y)$ to mean that x is y's mother, we simply write $m(y)$ to mean y's mother. The symbol m is a function symbol: it takes one argument and returns the mother of that argument. Using m, the two sentences above have simpler encodings than they had using M:

$$\forall x\, (C(x) \to Y(x, m(x)))$$

now expresses that every child is younger than its mother. Note that we need only one variable rather than two. Representing that Andy and Paul have the same maternal grandmother is even simpler; it is written

$$m(m(a)) = m(m(p))$$

quite directly saying that Andy's maternal grandmother is the same person as Paul's maternal grandmother.

[1] We assume that we are talking about genetic mothers, not adopted mothers, step mothers etc.

One can always do without function symbols, by using a predicate symbol instead. However, it is usually neater to use function symbols whenever possible, because we get more compact encodings. However, function symbols can be used only in situations in which we want to denote a single object. Above, we rely on the fact that every individual has a uniquely defined mother, so that we can talk about x's mother without risking any ambiguity (for example, if x had no mother, or two mothers). For this reason, we cannot have a function symbol $b(\cdot)$ for 'brother'. It might not make sense to talk about x's brother, for x might not have any brothers, or he might have several. 'Brother' must be coded as a binary predicate.

To exemplify this point further, if Mary has several brothers, then the claim that 'Ann likes Mary's brother' is ambiguous. It might be that Ann likes one of Mary's brothers, which we would write as

$$\exists x\,(B(x,m) \land L(a,x))$$

where B and L mean 'is brother of' and 'likes,' and a and m mean Ann and Mary. This sentence says that there exists an x which is a brother of Mary and is liked by Ann. Alternatively, if Ann likes all of Mary's brothers, we write it as

$$\forall x\,(B(x,m) \to L(a,x))$$

saying that any x which is a brother of Mary is liked by Ann. Predicates should be used if a 'function' such as 'your youngest brother' does not always have a value.

Different function symbols may take different numbers of arguments. Functions may take zero arguments and are then called *constants*: a and p above are constants for Andy and Paul, respectively. In a domain involving students and the grades they get in different courses, one might have the binary function symbol $g(\cdot,\cdot)$ taking two arguments: $g(x,y)$ refers to the grade obtained by student x in course y.

2.2 Predicate logic as a formal language

The discussion of the preceding section was intended to give an impression of how we code up sentences as formulas of predicate logic. In this section, we will be more precise about it, giving syntactic rules for the formation of predicate logic formulas. Because of the power of predicate logic, the language is much more complex than that of propositional logic.

The first thing to note is that there are two *sorts* of things involved in a predicate logic formula. The first sort denotes the objects that we are talking about: individuals such as a and p (referring to Andy and Paul) are examples, as are variables such as x and v. Function symbols also allow us to refer to objects: thus, $m(a)$ and $g(x, y)$ are also objects. Expressions in predicate logic which denote objects are called *terms*.

The other sort of things in predicate logic denotes truth values; expressions of this kind are *formulas*: $Y(x, m(x))$ is a formula, though x and $m(x)$ are terms.

A predicate vocabulary consists of three sets: a set of predicate symbols $\mathcal{P}$, a set of function symbols $\mathcal{F}$ and a set of constant symbols $\mathcal{C}$. Each predicate symbol and each function symbol comes with an arity, the number of arguments it expects. In fact, constants can be thought of as functions which don't take any arguments (and we even drop the argument brackets) – therefore, constants live in the set $\mathcal{F}$ together with the 'true' functions which do take arguments. From now on, we will drop the set $\mathcal{C}$, since it is convenient to do so, and stipulate that constants are 0-arity, so-called *nullary*, functions.

2.2.1 Terms

The terms of our language are made up of variables, constant symbols and functions applied to those. Functions may be nested, as in $m(m(x))$ or $g(m(a), c)$: the grade obtained by Andy's mother in the course c.

Definition 2.1 Terms are defined as follows.

- Any variable is a term.
- If $c \in \mathcal{F}$ is a nullary function, then c is a term.
- If $t_1, t_2, \ldots, t_n$ are terms and $f \in \mathcal{F}$ has arity $n > 0$, then $f(t_1, t_2, \ldots, t_n)$ is a term.
- Nothing else is a term.

In Backus Naur form we may write

$$t ::= x \;\mid\; c \;\mid\; f(t, \ldots, t)$$

where x ranges over a set of variables var, c over nullary function symbols in $\mathcal{F}$, and f over those elements of $\mathcal{F}$ with arity $n > 0$.

It is important to note that

- the first building blocks of terms are *constants* (nullary functions) and *variables*;
- more complex terms are built from function symbols using as many previously built terms as required by such function symbols; and
- the notion of terms is dependent on the set $\mathcal{F}$. If you change it, you change the set of terms.

Example 2.2 Suppose n, f and g are function symbols, respectively nullary, unary and binary. Then $g(f(n), n)$ and $f(g(n, f(n)))$ are terms, but $g(n)$ and $f(f(n), n)$ are not (they violate the arities). Suppose $0, 1, \ldots$ are nullary, s is unary, and $+$, $-$, and $*$ are binary. Then $*(-(2, +(s(x), y)), x)$ is a term, whose parse tree is illustrated in Figure 2.14 (page 159). Usually, the binary symbols are written infix rather than prefix; thus, the term is usually written $(2 - (s(x) + y)) * x$.

2.2.2 Formulas

The choice of sets $\mathcal{P}$ and $\mathcal{F}$ for predicate and function symbols, respectively, is driven by what we intend to describe. For example, if we work on a database representing relations between our kin we might want to consider $\mathcal{P} = \{M, F, S, D\}$, referring to *being male*, *being female*, *being a son of* ... and *being a daughter of* .... Naturally, F and M are unary predicates (they take one argument) whereas D and S are binary (taking two). Similarly, we may define $\mathcal{F} = \{mother\text{-}of, father\text{-}of\}$.

We already know what the terms over $\mathcal{F}$ are. Given that knowledge, we can now proceed to define the formulas of predicate logic.

Definition 2.3 We define the set of formulas over $(\mathcal{F}, \mathcal{P})$ inductively, using the already defined set of terms over $\mathcal{F}$:

- If $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, and if $t_1, t_2, \ldots, t_n$ are terms over $\mathcal{F}$, then $P(t_1, t_2, \ldots, t_n)$ is a formula.
- If ϕ is a formula, then so is $(\neg\phi)$.
- If ϕ and ψ are formulas, then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$ and $(\phi \to \psi)$.
- If ϕ is a formula and x is a variable, then $(\forall x\, \phi)$ and $(\exists x\, \phi)$ are formulas.
- Nothing else is a formula.

Note how the arguments given to predicates are always terms. This can also be seen in the Backus Naur form (BNF) for predicate logic:

$$\phi ::= P(t_1, t_2, \ldots, t_n) \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid (\forall x\, \phi) \mid (\exists x\, \phi) \tag{2.2}$$

where $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, t_i are terms over $\mathcal{F}$ and x is a variable. Recall that each occurrence of ϕ on the right-hand side of the ::= stands for any formula already constructed by these rules. (What role could predicate symbols of arity 0 play?)

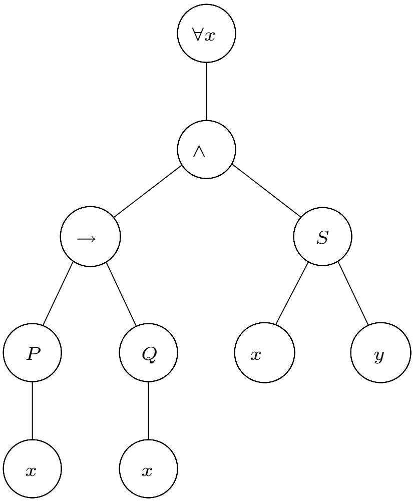


Figure 2.1. A parse tree of a predicate logic formula.

Convention 2.4 For convenience, we retain the usual binding priorities agreed upon in Convention 1.3 and add that $\forall y$ and $\exists y$ bind like $\neg$. Thus, the order is:

- $\neg$, $\forall y$ and $\exists y$ bind most tightly;
- then $\vee$ and $\wedge$;
- then $\rightarrow$, which is right-associative.

We also often omit brackets around quantifiers, provided that doing so introduces no ambiguities.

Predicate logic formulas can be represented by parse trees. For example, the parse tree in Figure 2.1 represents the formula $\forall x\,((P(x) \rightarrow Q(x)) \wedge S(x, y))$.

Example 2.5 Consider translating the sentence

Every son of my father is my brother.

into predicate logic. As before, the design choice is whether we represent 'father' as a predicate or as a function symbol.

1. As a predicate. We choose a constant m for 'me' or 'I,' so m is a term, and we choose further $\{S, F, B\}$ as the set of predicates with meanings

$$\begin{aligned} S(x,y): &\quad x \text{ is a son of } y \\ F(x,y): &\quad x \text{ is the father of } y \\ B(x,y): &\quad x \text{ is a brother of } y. \end{aligned}$$

Then the symbolic encoding of the sentence above is

$$\forall x\,\forall y\,(F(x,m) \wedge S(y,x) \to B(y,m)) \tag{2.3}$$

saying: 'For all x and all y, if x is a father of m and if y is a son of x, then y is a brother of m.'

2. As a function. We keep m, S and B as above and write f for the function which, given an argument, returns the corresponding father. Note that this works only because fathers are unique and always defined, so f really is a function as opposed to a mere relation.
The symbolic encoding of the sentence above is now

$$\forall x\,(S(x, f(m)) \to B(x,m)) \tag{2.4}$$

meaning: 'For all x, if x is a son of the father of m, then x is a brother of m;' it is less complex because it involves only one quantifier.

Formal specifications require *domain-specific knowledge.* Domain-experts often don't make some of this knowledge explicit, so a specifier may miss important constraints for a model or implementation. For example, the specification in (2.3) and (2.4) may seem right, but what about the case when the values of x and m are equal? If the domain of kinship is not common knowledge, then a specifier may not realize that a man cannot be his own brother. Thus, (2.3) and (2.4) are not completely correct!

2.2.3 Free and bound variables

The introduction of variables and quantifiers allows us to express the notions of *all* ... and *some* ... Intuitively, to verify that $\forall x\, Q(x)$ is true amounts to replacing x by any of its possible values and checking that Q holds for each one of them. There are two important and different senses in which such formulas can be 'true.' First, if we give concrete meanings to all predicate and function symbols involved we have a *model* and can *check* whether a formula is true for this particular model. For example, if a formula encodes a required behaviour of a hardware circuit, then we would want to know whether it is true for the model of the circuit. Second, one sometimes would like to ensure that certain formulas are true *for all models.* Consider $P(c) \wedge \forall y(P(y) \to Q(y)) \to Q(c)$ for a constant c; clearly, this formula should be true no matter what model we are looking at. It is this second kind of truth which is the primary focus of Section 2.3.

Unfortunately, things are more complicated if we want to define formally what it means for a formula to be true in a given model. Ideally, we seek a definition that we could use to write a computer program verifying that a formula holds in a given model. To begin with, we need to understand that variables occur in different ways. Consider the formula

$$\forall x\,((P(x) \to Q(x)) \land S(x, y)).$$

We draw its parse tree in the same way as for propositional formulas, but with two additional sorts of nodes:

- The quantifiers $\forall x$ and $\exists y$ form nodes and have, like negation, just one subtree.
- Predicate expressions, which are generally of the form $P(t_1, t_2, \dots, t_n)$, have the symbol P as a node, but now P has n many subtrees, namely the parse trees of the terms $t_1, t_2, \dots, t_n$.

So in our particular case above we arrive at the parse tree in Figure 2.1. You can see that variables occur at two different sorts of places. First, they appear next to quantifiers $\forall$ and $\exists$ in nodes like $\forall x$ and $\exists z$; such nodes always have one subtree, subsuming their scope to which the respective quantifier applies.

The other sort of occurrence of variables is *leaf nodes containing variables.* If variables are leaf nodes, then they stand for values that still have to be made concrete. There are two principal such occurrences:

1. In our example in Figure 2.1, we have three leaf nodes x. If we walk up the tree beginning at any one of these x leaves, we run into the quantifier $\forall x$. This means that those occurrences of x are actually *bound* to $\forall x$ so they represent, or stand for, *any possible value of* x.
2. In walking upwards, the only quantifier that the leaf node y runs into is $\forall x$ but that x has nothing to do with y; x and y are different place holders. So y is *free* in this formula. This means that its value has to be specified by some additional information, for example, the contents of a location in memory.

Definition 2.6 Let ϕ be a formula in predicate logic. An occurrence of x in ϕ is free in ϕ if it is a leaf node in the parse tree of ϕ such that there is no path upwards from that node x to a node $\forall x$ or $\exists x$. Otherwise, that occurrence of x is called bound. For $\forall x\,\phi$, or $\exists x\,\phi$, we say that ϕ – minus any of ϕ's subformulas $\exists x\,\psi$, or $\forall x\,\psi$ – is the scope of $\forall x$, respectively $\exists x$.

Thus, if x occurs in ϕ, then it is bound if, and only if, it is in the scope of some $\exists x$ or some $\forall x$; otherwise it is free. In terms of parse trees, the scope of a quantifier is just its subtree, minus any subtrees which re-introduce a

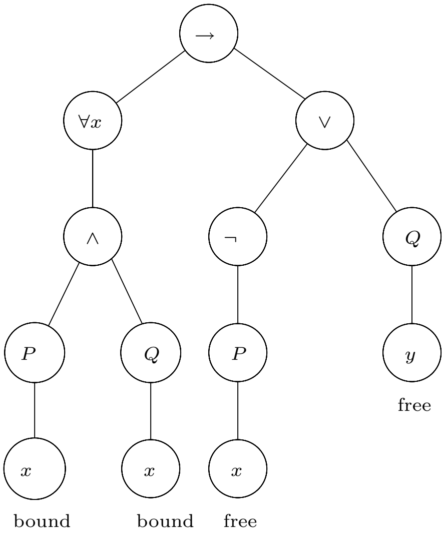


Figure 2.2. A parse tree of a predicate logic formula illustrating free and bound occurrences of variables.

quantifier for x; e.g. the scope of $\forall x$ in $\forall x\,(P(x) \to \exists x\,Q(x))$ is $P(x)$. It is quite possible, and common, that a variable is bound and free in a formula. Consider the formula

$$(\forall x\,(P(x) \land Q(x))) \to (\neg P(x) \lor Q(y))$$

and its parse tree in Figure 2.2. The two x leaves in the subtree of $\forall x$ are bound since they are in the scope of $\forall x$, but the leaf x in the right subtree of $\to$ is free since it is *not* in the scope of any quantifier $\forall x$ or $\exists x$. Note, however, that a single leaf either is under the scope of a quantifier, or it isn't. Hence *individual* occurrences of variables are either free or bound, never both at the same time.

2.2.4 Substitution

Variables are place holders so we must have some means of *replacing* them with more concrete information. On the syntactic side, we often need to replace a leaf node x by the parse tree of an entire term t. Recall from the definition of formulas that any replacement of x may only be a term; it could not be a predicate expression, or a more complex formula, for x serves as a term to a predicate symbol one step higher up in the parse tree (see Definition 2.1 and the grammar in (2.2)). In substituting t for x we have to leave untouched the *bound* leaves x since they are in the scope of some $\exists x$ or $\forall x$, i.e. they stand for *some unspecified* or *all* values respectively.

Definition 2.7 Given a variable x, a term t and a formula ϕ we define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

Substitutions are easily understood by looking at some examples. Let f be a function symbol with two arguments and ϕ the formula with the parse tree in Figure 2.1. Then $f(x, y)$ is a term and $\phi[f(x, y)/x]$ is just ϕ again. This is true because *all* occurrences of x are bound in ϕ, so *none* of them gets substituted.

Now consider ϕ to be the formula with the parse tree in Figure 2.2. Here we have one free occurrence of x in ϕ, so we substitute the parse tree of $f(x, y)$ for that free leaf node x and obtain the parse tree in Figure 2.3. Note that the bound x leaves are unaffected by this operation. You can see that the process of substitution is straightforward, but requires that it be applied *only to the free occurrences* of the variable to be substituted.

A word on notation: in writing $\phi[t/x]$, we really mean this to be the formula *obtained* by performing the operation $[t/x]$ on ϕ. Strictly speaking, the chain of symbols $\phi[t/x]$ is *not* a logical formula, but its *result* will be a formula, provided that ϕ was one in the first place.

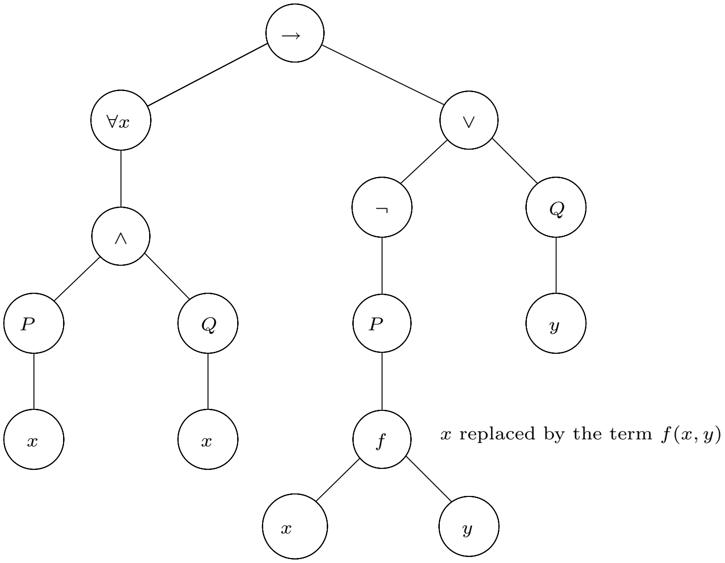


Figure 2.3. A parse tree of a formula resulting from substitution.

Unfortunately, substitutions can give rise to undesired side effects. In performing a substitution $\phi[t/x]$, the term t may contain a variable y, where free occurrences of x in ϕ are under the scope of $\exists y$ or $\forall y$ in ϕ. By carrying out this substitution $\phi[t/x]$, the value y, which might have been fixed by a concrete context, gets caught in the scope of $\exists y$ or $\forall y$. This binding capture overrides the context specification of the concrete value of y, for it will now stand for '*some unspecified*' or '*all*,' respectively. Such undesired variable captures are to be avoided at all costs.

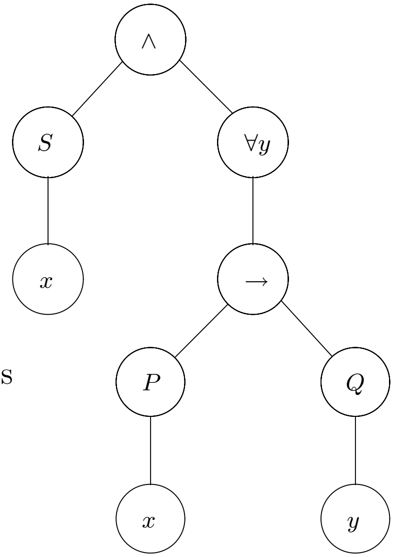
Definition 2.8 Given a term t, a variable x and a formula ϕ, we say that t is free for x in ϕ if no free x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

This definition is maybe hard to swallow. Let us think of it in terms of parse trees. Given the parse tree of ϕ and the parse tree of t, we can perform the substitution $[t/x]$ on ϕ to obtain the formula $\phi[t/x]$. The latter has a parse tree where all free x leaves of the parse tree of ϕ are replaced by the parse tree of t. What 't is free for x in ϕ' means is that the variable leaves of the parse tree of t won't become bound if placed into the bigger parse tree of $\phi[t/x]$. For example, if we consider x, t and ϕ in Figure 2.3, then t is free for x in ϕ since the *new* leaf variables x and y of t are not under the scope of any quantifiers involving x or y.

Example 2.9 Consider the ϕ with parse tree in Figure 2.4 and let t be $f(y, y)$. All two occurrences of x in ϕ are free. The leftmost occurrence of x could be substituted since it is not in the scope of any quantifier, but substituting the rightmost x leaf introduces a new variable y in t which becomes bound by $\forall y$. Therefore, $f(y, y)$ is not free for x in ϕ.

What if there are no free occurrences of x in ϕ? Inspecting the definition of 't is free for x in ϕ,' we see that *every* term t is free for x in ϕ in that case, since no free variable x of ϕ is below some quantifier in the parse tree of ϕ. So the problematic situation of variable capture in performing $\phi[t/x]$ cannot occur. Of course, in that case $\phi[t/x]$ is just ϕ again.

It might be helpful to compare 't is free for x in ϕ' with a precondition of calling a procedure for substitution. If you are asked to compute $\phi[t/x]$ in your exercises or exams, then that is what you should do; but any reasonable implementation of substitution used in a theorem prover would have to check whether t is free for x in ϕ and, if not, rename some variables with fresh ones to avoid the undesirable capture of variables.



the term $f(y, y)$ is not free for x in this formula

Figure 2.4. A parse tree for which a substitution has dire consequences.

2.3 Proof theory of predicate logic

2.3.1 Natural deduction rules

Proofs in the natural deduction calculus for predicate logic are similar to those for propositional logic in Chapter 1, except that we have new proof rules for dealing with the quantifiers and with the equality symbol. Strictly speaking, we are *overloading* the previously established proof rules for the propositional connectives ∧, ∨ etc. That simply means that any proof rule of Chapter 1 is still valid for logical formulas of predicate logic (we originally defined those rules for logical formulas of propositional logic). As in the natural deduction calculus for propositional logic, the additional rules for the quantifiers and equality will come in two flavours: introduction and elimination rules.

The proof rules for equality First, let us state the proof rules for equality. Here equality does not mean syntactic, or intensional, equality, but equality in terms of computation results. In either of these senses, any term t has to be equal to itself. This is expressed by the introduction rule for equality:

$$\frac{}{t = t}\;{=}\mathrm{i} \tag{2.5}$$

which is an axiom (as it does not depend on any premises). Notice that it may be invoked only if t is a term, our language doesn't permit us to talk about equality between formulas.

This rule is quite evidently sound, but it is not very useful on its own. What we need is a principle that allows us to substitute equals for equals repeatedly. For example, suppose that $y * (w + 2)$ equals $y * w + y * 2$; then it certainly must be the case that $z \geq y * (w + 2)$ implies $z \geq y * w + y * 2$ and vice versa. We may now express this substitution principle as the rule =e:

$$\frac{t_1 = t_2 \quad \phi[t_1/x]}{\phi[t_2/x]} = \mathrm{e}.$$

Note that t_1 and t_2 have to be free for x in ϕ, whenever we want to apply the rule =e; this is an example of a *side condition* of a proof rule.

Convention 2.10 Throughout this section, when we write a substitution in the form $\phi[t/x]$, we implicitly assume that t is free for x in ϕ; for, as we saw in the last section, a substitution doesn't make sense otherwise.

We obtain proof

1	$(x + 1) = (1 + x)$	premise
2	$(x + 1 > 1) \to (x + 1 > 0)$	premise
3	$(1 + x > 1) \to (1 + x > 0)$	$=$e $1, 2$

establishing the validity of the sequent

$$x + 1 = 1 + x, \, (x + 1 > 1) \to (x + 1 > 0) \;\vdash\; (1 + x) > 1 \to (1 + x) > 0.$$

In this particular proof t_1 is $(x + 1)$, t_2 is $(1 + x)$ and ϕ is $(x > 1) \to (x > 0)$. We used the name =e since it reflects what this rule is doing to data: it eliminates the equality in $t_1 = t_2$ by replacing all t_1 in $\phi[t_1/x]$ with t_2. This is a sound substitution principle, since the assumption that t_1 equals t_2 guarantees that the logical meanings of $\phi[t_1/x]$ and $\phi[t_2/x]$ match.

The principle of substitution, in the guise of the rule =e, is quite powerful. Together with the rule =i, it allows us to show the sequents

$$t_1 = t_2 \vdash t_2 = t_1 \tag{2.6}$$

$$t_1 = t_2, \; t_2 = t_3 \vdash t_1 = t_3. \tag{2.7}$$

A proof for (2.6) is:

1	$t_1 = t_2$	premise
2	$t_1 = t_1$	=i
3	$t_2 = t_1$	=e $1, 2$

where ϕ is $x = t_1$. A proof for (2.7) is:

1	$t_2 = t_3$	premise
2	$t_1 = t_2$	premise
3	$t_1 = t_3$	=e $1, 2$

where ϕ is $t_1 = x$, so in line 2 we have $\phi[t_2/x]$ and in line 3 we obtain $\phi[t_3/x]$, as given by the rule =e applied to lines 1 and 2. Notice how we applied the scheme =e with several different instantiations.

Our discussion of the rules =i and =e has shown that they force equality to be *reflexive* (2.5), *symmetric* (2.6) and *transitive* (2.7). These are minimal and necessary requirements for any sane concept of (extensional) equality. We leave the topic of equality for now to move on to the proof rules for quantifiers.

The proof rules for universal quantification The rule for eliminating ∀ is the following:

$$\frac{\forall x\, \phi}{\phi[t/x]}\ \forall x\, \mathrm{e}.$$

It says: If $\forall x\, \phi$ is true, then you could replace the x in ϕ by any term t (given, as usual, the side condition that t be free for x in ϕ) and conclude that $\phi[t/x]$ is true as well. The intuitive soundness of this rule is self-evident.

Recall that $\phi[t/x]$ is obtained by replacing all free occurrences of x in ϕ by t. You may think of the term t as a more concrete *instance* of x. Since ϕ is assumed to be true for all x, that should also be the case for any term t.

Example 2.11 To see the necessity of the proviso that t be free for x in ϕ, consider the case that ϕ is $\exists y\, (x < y)$ and the term to be substituted for x is y. Let's suppose we are reasoning about numbers with the usual 'smaller than' relation. The statement $\forall x\, \phi$ then says that for all numbers n there is some bigger number m, which is indeed true of integers or real numbers. However, $\phi[y/x]$ is the formula $\exists y\, (y < y)$ saying that there is a number which is bigger than itself. This is wrong; and we must not allow a proof rule which derives semantically wrong things from semantically valid ones. Clearly, what went wrong was that y became bound in the process of substitution; y is not free for x in ϕ. Thus, in going from $\forall x\,\phi$ to $\phi[t/x]$, we have to enforce the side condition that t be free for x in ϕ: use a fresh variable for y to change ϕ to, say, $\exists z\,(x < z)$ and then apply $[y/x]$ to that formula, rendering $\exists z\,(y < z)$.

The rule $\forall x\,$i is a bit more complicated. It employs a proof box similar to those we have already seen in natural deduction for propositional logic, but this time the box is to stipulate the scope of the 'dummy variable' x_0 rather than the scope of an assumption. The rule $\forall x\,$i is written

$$\frac{\boxed{\begin{array}{ll} x_0 & \\ & \vdots \\ & \phi[x_0/x] \end{array}}}{\forall x\,\phi}\ \forall x\,\mathrm{i}.$$

It says: If, starting with a 'fresh' variable x_0, you are able to prove some formula $\phi[x_0/x]$ with x_0 in it, then (*because x_0 is fresh*) you can derive $\forall x\,\phi$. The important point is that x_0 is a new variable which doesn't occur *anywhere outside its box*; we think of it as an *arbitrary* term. Since we assumed nothing about this x_0, anything would work in its place; hence the conclusion $\forall x\,\phi$.

It takes a while to understand this rule, since it seems to be going from the particular case of ϕ to the general case $\forall x\,\phi$. The side condition, that x_0 does not occur outside the box, is what allows us to get away with this.

To understand this, think of the following analogy. If you want to prove to someone that you can, say, split a tennis ball in your hand by squashing it, you might say 'OK, give me a tennis ball and I'll split it.' So we give you one and you do it. But how can we be sure that you could split *any* tennis ball in this way? Of course, we can't give you *all of them*, so how could we be sure that you could split any one? Well, we assume that the one you did split was an arbitrary, or 'random,' one, i.e. that it wasn't special in any way – like a ball which you may have 'prepared' beforehand; and that is enough to convince us that you could split *any* tennis ball. Our rule says that if you can prove ϕ about an x_0 that isn't special in any way, then you could prove it for any x whatsoever.

To put it another way, the step from ϕ to $\forall x\,\phi$ is legitimate only if we have arrived at ϕ in such a way that none of its assumptions contain x as a free variable. Any assumption which has a free occurrence of x puts constraints on such an x. For example, the assumption bird(x) confines x to the realm of birds and anything we can prove about x using this formula will have to be a statement restricted to birds and not about anything else we might have had in mind.

It is time we looked at an example of these proof rules at work. Here is a proof of the sequent $\forall x\,(P(x) \to Q(x)),\ \forall x\,P(x)\ \vdash\ \forall x\,Q(x)$:

1		$\forall x\,(P(x) \to Q(x))$	premise
2		$\forall x\,P(x)$	premise
3	x_0	$P(x_0) \to Q(x_0)$	$\forall x\,$e 1
4		$P(x_0)$	$\forall x\,$e 2
5		$Q(x_0)$	$\to$e 3, 4
6		$\forall x\,Q(x)$	$\forall x\,$i 3–5

The structure of this proof is guided by the fact that the conclusion is a $\forall$ formula. To arrive at this, we will need an application of $\forall x\,$i, so we set up the box controlling the scope of x_0. The rest is now mechanical: we prove $\forall x\,Q(x)$ by proving $Q(x_0)$; but the latter we can prove as soon as we can prove $P(x_0)$ and $P(x_0) \to Q(x_0)$, which themselves are instances of the premises (obtained by $\forall$e with the term x_0). Note that we wrote the name of the dummy variable to the left of the first proof line in its scope box.

Here is a simpler example which uses only $\forall x\,$e: we show the validity of the sequent $P(t),\ \forall x\,(P(x) \to \neg Q(x))\ \vdash\ \neg Q(t)$ for any term t:

1	$P(t)$	premise
2	$\forall x\,(P(x) \to \neg Q(x))$	premise
3	$P(t) \to \neg Q(t)$	$\forall x\,$e 2
4	$\neg Q(t)$	$\to$e 3, 1

Note that we invoked $\forall x\,$e with the same instance t as in the assumption $P(t)$. If we had invoked $\forall x\,$e with y, say, and obtained $P(y) \to \neg Q(y)$, then that would have been valid, but it would not have been helpful in the case that y was different from t. Thus, $\forall x\,$e is really a *scheme* of rules, one for each term t (free for x in ϕ), and we should make our choice on the basis of consistent pattern matching. Further, note that we have rules $\forall x\,$i and $\forall x\,$e *for each variable* x. In particular, there are rules $\forall y\,$i, $\forall y\,$e and so on. We will write ∀i and ∀e when we speak about such rules without concern for the actual quantifier variable.

Notice also that, although the square brackets representing substitution appear in the rules ∀i and ∀e, they do not appear when we use those rules. The reason for this is that we actually carry out the substitution that is asked for. In the rules, the expression $\phi[t/x]$ means: 'ϕ, but with free occurrences of x replaced by t.' Thus, if ϕ is $P(x, y) \to Q(y, z)$ and the rule refers to $\phi[a/y]$, we carry out the substitution and write $P(x, a) \to Q(a, z)$ in the proof.

A helpful way of understanding the universal quantifier rules is to compare the rules for ∀ with those for ∧. The rules for ∀ are in some sense generalisations of those for ∧; whereas ∧ has just two conjuncts, ∀ acts like it conjoins lots of formulas (one for each substitution instance of its variable). Thus, whereas ∧i has two premises, $\forall x$ i has a premise $\phi[x_0/x]$ for each possible 'value' of x_0. Similarly, where and-elimination allows you to deduce from $\phi \wedge \psi$ whichever of ϕ and ψ you like, forall-elimination allows you to deduce $\phi[t/x]$ from $\forall x\, \phi$, for whichever t you (and the side condition) like. To say the same thing another way: think of $\forall x$ i as saying: to prove $\forall x\, \phi$, you have to prove $\phi[x_0/x]$ for every possible value x_0; while ∧i says that to prove $\phi_1 \wedge \phi_2$ you have to prove ϕ_i for every $i = 1, 2$.

The proof rules for existential quantification The analogy between ∀ and ∧ extends also to ∃ and ∨; and you could even try to guess the rules for ∃ by starting from the rules for ∨ and applying the same ideas as those that related ∧ to ∀. For example, we saw that the rules for or-introduction were a sort of dual of those for and-elimination; to emphasise this point, we could write them as

$$\frac{\phi_1 \wedge \phi_2}{\phi_k} \wedge\mathrm{e_k} \qquad \frac{\phi_k}{\phi_1 \vee \phi_2} \vee\mathrm{i_k}$$

where k can be chosen to be either 1 or 2. Therefore, given the form of forall-elimination, we can infer that exists-introduction must be simply

$$\frac{\phi[t/x]}{\exists x \phi} \exists x\,\mathrm{i}.$$

Indeed, this is correct: it simply says that we can deduce $\exists x\, \phi$ whenever we have $\phi[t/x]$ for some term t (naturally, we impose the side condition that t be free for x in ϕ).

In the rule ∃i, we see that the formula $\phi[t/x]$ contains, from a computational point of view, more information than $\exists x\, \phi$. The latter merely says that ϕ holds for some, unspecified, value of x; whereas $\phi[t/x]$ has a witness t at its disposal. Recall that the square-bracket notation asks us actually to carry out the substitution. However, the notation $\phi[t/x]$ is somewhat misleading since it suggests not only the right witness t but also the formula ϕ itself. For example, consider the situation in which t equals y such that $\phi[y/x]$ is $y = y$. Then you can check for yourself that ϕ could be a number of things, like $x = x$ or $x = y$. Thus, $\exists x\, \phi$ will depend on which of these ϕ you were thinking of.

Extending the analogy between $\exists$ and $\vee$, the rule $\vee$e leads us to the following formulation of $\exists$e:

$$\dfrac{\exists x\,\phi \qquad \boxed{\begin{array}{c} x_0 \;\; \phi[x_0/x] \\ \vdots \\ \chi \end{array}}}{\chi}\ \exists\mathrm{e}.$$

Like $\vee$e, it involves a case analysis. The reasoning goes: We know $\exists x\, \phi$ is true, so ϕ is true for at least one 'value' of x. So we do a case analysis over all those possible values, writing x_0 as a generic value representing them all. If assuming $\phi[x_0/x]$ allows us to prove some χ which doesn't mention x_0, then this χ must be true whichever x_0 makes $\phi[x_0/x]$ true. And that's precisely what the rule $\exists$e allows us to deduce. Of course, we impose the side condition that x_0 can't occur outside its box (therefore, in particular, it cannot occur in χ). The box is controlling two things: the scope of x_0 and also the scope of the assumption $\phi[x_0/x]$.

Just as $\vee$e says that to use $\phi_1 \vee \phi_2$, you have to be prepared for either of the ϕ_i, so $\exists$e says that to use $\exists x\, \phi$ you have to be prepared for any possible $\phi[x_0/x]$. Another way of thinking about $\exists$e goes like this: If you know $\exists x\, \phi$ and you can derive some χ from $\phi[x_0/x]$, i.e. by giving a name to the thing you know exists, then you can derive χ even without giving that thing a name (provided that χ does not refer to the name x_0).

The rule $\exists x\,$e is also similar to $\vee$e in the sense that both of them are elimination rules which don't have to conclude a *subformula* of the formula they are about to eliminate. Please verify that all other elimination rules introduced so far have this *subformula property.*[2] This property is computationally very pleasant, for it allows us to narrow down the search space for a proof dramatically. Unfortunately, $\exists x\,$e, like its cousin $\vee$e, is not of that computationally benign kind.

[2] For $\forall x\,$e we perform a substitution $[t/x]$, but it preserves the logical structure of ϕ.

Let us practice these rules on a couple of examples. Certainly, we should be able to prove the validity of the sequent $\forall x\, \phi \vdash \exists x\, \phi$. The proof

1	$\forall x\, \phi$	premise
2	$\phi[x/x]$	$\forall x\,$e 1
3	$\exists x\, \phi$	$\exists x\,$i 2

demonstrates that, where we chose t to be x with respect to both $\forall x\,$e and to $\exists x\,$i (and note that x is free for x in ϕ and that $\phi[x/x]$ is simply ϕ again).

Proving the validity of the sequent $\forall x\, (P(x) \to Q(x)),\ \exists x\, P(x) \vdash \exists x\, Q(x)$ is more complicated:

1		$\forall x\, (P(x) \to Q(x))$	premise
2		$\exists x\, P(x)$	premise
3	x_0	$P(x_0)$	assumption
4		$P(x_0) \to Q(x_0)$	$\forall x\,$e 1
5		$Q(x_0)$	$\to$e $4, 3$
6		$\exists x\, Q(x)$	$\exists x\,$i 5
7		$\exists x\, Q(x)$	$\exists x\,$e $2, 3{-}6$

The motivation for introducing the box in line 3 of this proof is the existential quantifier in the premise $\exists x\, P(x)$ which has to be eliminated. Notice that the $\exists$ in the conclusion has to be introduced *within the box* and observe the nesting of these two steps. The formula $\exists x\, Q(x)$ in line 6 is the instantiation of χ in the rule $\exists$e and does not contain an occurrence of x_0, so it is allowed to leave the box to line 7. The almost identical 'proof'

1		$\forall x\, (P(x) \to Q(x))$	premise
2		$\exists x\, P(x)$	premise
3	x_0	$P(x_0)$	assumption
4		$P(x_0) \to Q(x_0)$	$\forall x\,$e 1
5		$Q(x_0)$	$\to$e $4, 3$
6		$Q(x_0)$	$\exists x\,$e $2, 3{-}5$
7		$\exists x\, Q(x)$	$\exists x\,$i 6

is illegal! Line 6 allows the fresh parameter x_0 to escape the scope of the box which declares it. This is not permissible and we will see on page 116 an example where such illicit use of proof rules results in unsound arguments.

A sequent with a slightly more complex proof is

$$\forall x\,(Q(x) \to R(x)),\ \exists x\,(P(x) \wedge Q(x))\ \vdash\ \exists x\,(P(x) \wedge R(x))$$

and could model some argument such as

> *If all quakers are reformists and if there is a protestant who is also a quaker, then there must be a protestant who is also a reformist.*

One possible proof strategy is to assume $P(x_0) \wedge Q(x_0)$, get the instance $Q(x_0) \to R(x_0)$ from $\forall x\,(Q(x) \to R(x))$ and use $\wedge\mathrm{e}_2$ to get our hands on $Q(x_0)$, which gives us $R(x_0)$ via $\to$e . . . :

1		$\forall x\,(Q(x) \to R(x))$	premise
2		$\exists x\,(P(x) \wedge Q(x))$	premise
3	x_0	$P(x_0) \wedge Q(x_0)$	assumption
4		$Q(x_0) \to R(x_0)$	$\forall x\,\mathrm{e}\ 1$
5		$Q(x_0)$	$\wedge\mathrm{e}_2\ 3$
6		$R(x_0)$	$\to\mathrm{e}\ 4, 5$
7		$P(x_0)$	$\wedge\mathrm{e}_1\ 3$
8		$P(x_0) \wedge R(x_0)$	$\wedge\mathrm{i}\ 7, 6$
9		$\exists x\,(P(x) \wedge R(x))$	$\exists x\,\mathrm{i}\ 8$
10		$\exists x\,(P(x) \wedge R(x))$	$\exists x\,\mathrm{e}\ 2, 3{-}9$

Note the strategy of this proof: We list the two premises. The second premise is of use here only if we apply $\exists x\,$e to it. This sets up the proof box in lines 3−9 as well as the fresh parameter name x_0. Since we want to prove $\exists x\,(P(x) \wedge R(x))$, this formula has to be the last one in the box (our goal) and the rest involves $\forall x\,$e and $\exists x\,$i.

The rules $\forall$i and $\exists$e both have the side condition that the dummy variable cannot occur outside the box in the rule. Of course, these rules may still be nested, by choosing another fresh name (e.g. y_0) for the dummy variable. For example, consider the sequent $\exists x\, P(x), \forall x\, \forall y\, (P(x) \to Q(y))\ \vdash\ \forall y\, Q(y)$. (Look how strong the second premise is, by the way: given any x, y, if $P(x)$, then $Q(y)$. This means that, if there is any object with the property P, then all objects shall have the property Q.) Its proof goes as follows: We take an arbitrary y_0 and prove $Q(y_0)$; this we do by observing that, since some x

satisfies P, so by the second premise any y satisfies Q:

1		$\exists x\, P(x)$	premise
2		$\forall x \forall y\, (P(x) \to Q(y))$	premise
3	y_0		
4	x_0	$P(x_0)$	assumption
5		$\forall y\, (P(x_0) \to Q(y))$	$\forall x$ e 2
6		$P(x_0) \to Q(y_0)$	$\forall y$ e 5
7		$Q(y_0)$	$\to$e 6, 4
8		$Q(y_0)$	$\exists x$ e 1, 4–7
9		$\forall y\, Q(y)$	$\forall y$ i 3–8

There is no special reason for picking x_0 as a name for the dummy variable we use for $\forall x$ and $\exists x$ and y_0 as a name for $\forall y$ and $\exists y$. We do this only because it makes it easier for us humans. Again, study the strategy of this proof. We ultimately have to show a $\forall y$ formula which requires us to use $\forall y$ i, i.e. we need to open up a proof box (lines 3–8) whose subgoal is to prove a generic instance $Q(y_0)$. Within that box we want to make use of the premise $\exists x\, P(x)$ which results in the proof box set-up of lines 4–7. Notice that, in line 8, we may well move $Q(y_0)$ out of the box controlled by x_0.

We have repeatedly emphasised the point that the dummy variables in the rules $\exists$e and $\forall$i must not occur outside their boxes. Here is an example which shows how things would go wrong if we didn't have this side condition. Consider the invalid sequent $\exists x\, P(x), \forall x\, (P(x) \to Q(x)) \vdash \forall y\, Q(y)$. (Compare it with the previous sequent; the second premise is now much weaker, allowing us to conclude Q only for those objects for which we know P.) Here is an alleged 'proof' of its validity:

1		$\exists x\, P(x)$	premise
2		$\forall x\, (P(x) \to Q(x))$	premise
3	x_0		
4	x_0	$P(x_0)$	assumption
5		$P(x_0) \to Q(x_0)$	$\forall x$ e 2
6		$Q(x_0)$	$\to$e 5, 4
7		$Q(x_0)$	$\exists x$ e 1, 4–6
8		$\forall y\, Q(y)$	$\forall y$ i 3–7

The last step introducing $\forall y$ is *not* the bad one; that step is fine. The bad one is the second from last one, concluding $Q(x_0)$ by $\exists x\,\mathrm{e}$ and violating the side condition that x_0 may not leave the scope of its box. You can try a few other ways of 'proving' this sequent, but none of them should work (assuming that our proof system is sound with respect to semantic entailment, which we define in the next section). Without this side condition, we would also be able to prove that 'all x satisfy the property P as soon as one of them does so,' a semantic disaster of biblical proportions!

2.3.2 Quantifier equivalences

We have already hinted at semantic equivalences between certain forms of quantification. Now we want to provide formal proofs for some of the most commonly used quantifier equivalences. Quite a few of them involve several quantifications over more than just one variable. Thus, this topic is also good practice for using the proof rules for quantifiers in a nested fashion.

For example, the formula $\forall x\, \forall y\, \phi$ should be equivalent to $\forall y\, \forall x\, \phi$ since both say that ϕ should hold for all values of x and y. What about $(\forall x\, \phi) \wedge (\forall x\, \psi)$ versus $\forall x\, (\phi \wedge \psi)$? A moment's thought reveals that they should have the same meaning as well. But what if the second conjunct does not start with $\forall x$? So what if we are looking at $(\forall x\, \phi) \wedge \psi$ in general and want to compare it with $\forall x\, (\phi \wedge \psi)$? Here we need to be careful, since x might be free in ψ and would then become bound in the formula $\forall x\, (\phi \wedge \psi)$.

Example 2.12 We may specify 'Not all birds can fly.' as $\neg\forall x\, (B(x) \to F(x))$ or as $\exists x\, (B(x) \wedge \neg F(x))$. The former formal specification is closer to the structure of the English specification, but the latter is logically equivalent to the former. Quantifier equivalences help us in establishing that specifications that 'look' different are really saying the same thing.

Here are some quantifier equivalences which you should become familiar with. As in Chapter 1, we write $\phi_1 \dashv\vdash \phi_2$ as an abbreviation for the validity of $\phi_1 \vdash \phi_2$ and $\phi_2 \vdash \phi_1$.

Theorem 2.13 Let ϕ and ψ be formulas of predicate logic. Then we have the following equivalences:

1. (a) $\neg\forall x\, \phi \dashv\vdash \exists x\, \neg\phi$
   (b) $\neg\exists x\, \phi \dashv\vdash \forall x\, \neg\phi$.
2. Assuming that x is not free in ψ:

(a) $\forall x\, \phi \wedge \psi \dashv\vdash \forall x\, (\phi \wedge \psi)$[3]
(b) $\forall x\, \phi \vee \psi \dashv\vdash \forall x\, (\phi \vee \psi)$
(c) $\exists x\, \phi \wedge \psi \dashv\vdash \exists x\, (\phi \wedge \psi)$
(d) $\exists x\, \phi \vee \psi \dashv\vdash \exists x\, (\phi \vee \psi)$
(e) $\forall x\, (\psi \to \phi) \dashv\vdash \psi \to \forall x\, \phi$
(f) $\exists x\, (\phi \to \psi) \dashv\vdash \forall x\, \phi \to \psi$
(g) $\forall x\, (\phi \to \psi) \dashv\vdash \exists x\, \phi \to \psi$
(h) $\exists x\, (\psi \to \phi) \dashv\vdash \psi \to \exists x\, \phi$.

3. (a) $\forall x\, \phi \wedge \forall x\, \psi \dashv\vdash \forall x\, (\phi \wedge \psi)$
   (b) $\exists x\, \phi \vee \exists x\, \psi \dashv\vdash \exists x\, (\phi \vee \psi)$.
4. (a) $\forall x\, \forall y\, \phi \dashv\vdash \forall y\, \forall x\, \phi$
   (b) $\exists x\, \exists y\, \phi \dashv\vdash \exists y\, \exists x\, \phi$.

PROOF: We will prove most of these sequents; the proofs for the remaining ones are straightforward adaptations and are left as exercises. Recall that we sometimes write $\perp$ to denote any contradiction.

1. (a) We will lead up to this by proving the validity of two simpler sequents first: $\neg(p_1 \wedge p_2) \vdash \neg p_1 \vee \neg p_2$ and then $\neg \forall x\, P(x) \vdash \exists x\, \neg P(x)$. The reason for proving the first of these is to illustrate the close relationship between $\wedge$ and $\vee$ on the one hand and $\forall$ and $\exists$ on the other – think of a model with just two elements 1 and 2 such that p_i $(i = 1, 2)$ stands for $P(x)$ evaluated at i. The idea is that proving this propositional sequent should give us inspiration for proving the second one of predicate logic. The reason for proving the latter sequent is that it is a special case (in which ϕ equals $P(x)$) of the one we are really after, so again it should be simpler while providing some inspiration. So, let's go.

1	$\neg(p_1 \wedge p_2)$			premise
2	$\neg(\neg p_1 \vee \neg p_2)$			assumption
3	$\neg p_1$	assumption	$\neg p_2$	assumption
4	$\neg p_1 \vee \neg p_2$	$\vee\mathrm{i}_1$ 3	$\neg p_1 \vee \neg p_2$	$\vee\mathrm{i}_2$ 3
5	$\perp$	$\neg$e 4, 2	$\perp$	$\neg$e 4, 2
6	p_1	PBC 3−5	p_2	PBC 3−5
7	$p_1 \wedge p_2$			$\wedge$i 6, 6
8	$\perp$			$\neg$e 7, 1
9	$\neg p_1 \vee \neg p_2$			PBC 2−8

[3] Remember that $\forall x\, \phi \wedge \psi$ is implicitly bracketed as $(\forall x\, \phi) \wedge \psi$, by virtue of the binding priorities.

You have seen this sort of proof before, in Chapter 1. It is an example of something which requires proof by contradiction, or $\neg\neg$e, or LEM (meaning that it simply cannot be proved in the reduced natural deduction system which discards these three rules) – in fact, the proof above used the rule PBC three times.

Now we prove the validity of $\neg\forall x\, P(x) \vdash \exists x\, \neg P(x)$ similarly, except that where the rules for $\wedge$ and $\vee$ were used we now use those for $\forall$ and $\exists$:

1	$\neg\forall x\, P(x)$	premise
2	$\neg\exists x\, \neg P(x)$	assumption
3	x_0	
4	$\neg P(x_0)$	assumption
5	$\exists x \neg P(x)$	$\exists x$ i 4
6	$\bot$	$\neg$e $5, 2$
7	$P(x_0)$	PBC 4−6
8	$\forall x\, P(x)$	$\forall x$ i 3−7
9	$\bot$	$\neg$e $8, 1$
10	$\exists x\, \neg P(x)$	PBC 2−9

You will really benefit by spending time understanding the way this proof mimics the one above it. This insight is very useful for constructing predicate logic proofs: you first construct a similar propositional proof and then mimic it.

Next we prove that $\neg\forall x\, \phi \vdash \exists x\, \neg\phi$ is valid:

1	$\neg\forall x\, \phi$	premise
2	$\neg\exists x\, \neg\phi$	assumption
3	x_0	
4	$\neg\phi[x_0/x]$	assumption
5	$\exists x \neg\phi$	$\exists x$ i 4
6	$\bot$	$\neg$e $5, 2$
7	$\phi[x_0/x]$	PBC 4−6
8	$\forall x\, \phi$	$\forall x$ i 3−7
9	$\bot$	$\neg$e $8, 1$
10	$\exists x\, \neg\phi$	PBC 2−9

Proving that the reverse $\exists x\,\neg\phi \vdash \neg\forall x\,\phi$ is valid is more straightforward, for it does not involve proof by contradiction, $\neg\neg$e, or LEM. Unlike its converse, it has a constructive proof which the intuitionists do accept. We could again prove the corresponding propositional sequent, but we leave that as an exercise.

1		$\exists x\,\neg\phi$	assumption
2		$\forall x\,\phi$	assumption
3	x_0		
4		$\neg\phi[x_0/x]$	assumption
5		$\phi[x_0/x]$	$\forall x\,$e 2
6		$\bot$	$\neg$e $5, 4$
7		$\bot$	$\exists x\,$e $1, 3{-}6$
8		$\neg\forall x\,\phi$	$\neg$i $2{-}7$

2. (a) Validity of $\forall x\,\phi \wedge \psi \vdash \forall x\,(\phi \wedge \psi)$ can be proved thus:

1		$(\forall x\,\phi) \wedge \psi$	premise
2		$\forall x\,\phi$	$\wedge$e$_1$ 1
3		ψ	$\wedge$e$_2$ 1
4	x_0		
5		$\phi[x_0/x]$	$\forall x\,$e 2
6		$\phi[x_0/x] \wedge \psi$	$\wedge$i $5, 3$
7		$(\phi \wedge \psi)[x_0/x]$	identical to 6, since x not free in ψ
8		$\forall x\,(\phi \wedge \psi)$	$\forall x\,$i $4{-}7$

The argument for the reverse validity can go like this:

1		$\forall x\,(\phi \wedge \psi)$	premise
2	x_0		
3		$(\phi \wedge \psi)[x_0/x]$	$\forall x\,$e 1
4		$\phi[x_0/x] \wedge \psi$	identical to 3, since x not free in ψ
5		ψ	$\wedge$e$_2$ 3
6		$\phi[x_0/x]$	$\wedge$e$_1$ 3
7		$\forall x\,\phi$	$\forall x\,$i $2{-}6$
8		$(\forall x\,\phi) \wedge \psi$	$\wedge$i $7, 5$

Notice that the use of $\wedge$i in the last line is permissible, because ψ was obtained for any instantiation of the formula in line 1; although a formal tool for proof support may complain about such practice.

3. (b) The sequent $(\exists x\,\phi) \vee (\exists x\,\psi) \vdash \exists x\,(\phi \vee \psi)$ is proved valid using the rule $\vee$e; so we have two principal cases, each of which requires the rule $\exists x\,$i:

1		$(\exists x\,\phi) \vee (\exists x\,\psi)$			premise
2		$\exists x\,\phi$		$\exists x\,\psi$	assumpt.
3	x_0	$\phi[x_0/x]$	x_0	$\psi[x_0/x]$	assumpt.
4		$\phi[x_0/x]\vee\psi[x_0/x]$		$\phi[x_0/x]\vee\psi[x_0/x]$	$\vee$i 3
5		$(\phi \vee \psi)[x_0/x]$		$(\phi \vee \psi)[x_0/x]$	identical
6		$\exists x\,(\phi \vee \psi)$		$\exists x\,(\phi \vee \psi)$	$\exists x\,$i 5
7		$\exists x\,(\phi \vee \psi)$		$\exists x\,(\phi \vee \psi)$	$\exists x\,$e 2, 3–6
8		$\exists x\,(\phi \vee \psi)$			$\vee$e 1, 2–7

The converse sequent has $\exists x\,(\phi \vee \psi)$ as premise, so its proof has to use $\exists x\,$e as its last rule; for that rule, we need $\phi \vee \psi$ as a temporary assumption and need to conclude $(\exists x\,\phi) \vee (\exists x\,\psi)$ from those data; of course, the assumption $\phi \vee \psi$ requires the usual case analysis:

1		$\exists x\,(\phi \vee \psi)$		premise
2	x_0	$(\phi \vee \psi)[x_0/x]$		assumption
3		$\phi[x_0/x] \vee \psi[x_0/x]$		identical
4		$\phi[x_0/x]$	$\psi[x_0/x]$	assumption
5		$\exists x\,\phi$	$\exists x\,\psi$	$\exists x\,$i 4
6		$\exists x\,\phi \vee \exists x\,\psi$	$\exists x\,\phi \vee \exists x\,\psi$	$\vee$i 5
7		$\exists x\,\phi \vee \exists x\,\psi$		$\vee$e 3, 4–6
8		$\exists x\,\phi \vee \exists x\,\psi$		$\exists x\,$e 1, 2–7

4. (b) Given the premise $\exists x\,\exists y\,\phi$, we have to nest $\exists x\,$e and $\exists y\,$e to conclude $\exists y\,\exists x\,\phi$. Of course, we have to obey the format of these elimination rules as done below:

1		$\exists x\, \exists y\, \phi$	premise
2	x_0	$(\exists y\, \phi)[x_0/x]$	assumption
3		$\exists y\, (\phi[x_0/x])$	identical, since x, y different variables
4	y_0	$\phi[x_0/x][y_0/y]$	assumption
5		$\phi[y_0/y][x_0/x]$	identical, since x, y, x_0, y_0 different variables
6		$\exists x\, \phi[y_0/y]$	$\forall x$ i 5
7		$\exists y\, \exists x\, \phi$	$\forall y$ i 6
8		$\exists y\, \exists x\, \phi$	$\exists y$ e3, 4−7
9		$\exists y\, \exists x\, \phi$	$\exists x$ e1, 2−8

The validity of the converse sequent is proved in the same way by swapping the roles of x and y. □

2.4 Semantics of predicate logic

Having seen how natural deduction of propositional logic can be extended to predicate logic, let's now look at how the semantics of predicate logic works. Just like in the propositional case, the semantics should provide a separate, but ultimately equivalent, characterisation of the logic. By 'separate,' we mean that the meaning of the connectives is defined in a different way; in proof theory, they were defined by proof rules providing an *operative* explanation. In semantics, we expect something like truth tables. By 'equivalent,' we mean that we should be able to prove soundness and completeness, as we did for propositional logic – although a fully fledged proof of soundness and completeness for predicate logic is beyond the scope of this book.

Before we begin describing the semantics of predicate logic, let us look more closely at the real difference between a semantic and a proof-theoretic account. In proof theory, the basic object which is constructed is a proof. Let us write Γ as a shorthand for lists of formulas $\phi_1, \phi_2, \ldots, \phi_n$. Thus, to show that $\Gamma \vdash \psi$ is valid, we need to provide a proof of ψ from Γ. Yet, how can we show that ψ is not a consequence of Γ? Intuitively, this is harder; how can you possibly show that *there is no proof* of something? You would have to consider every 'candidate' proof and show it is not one. Thus, proof theory gives a 'positive' characterisation of the logic; it provides convincing evidence for assertions like '$\Gamma \vdash \psi$ is valid,' but it is not very useful for establishing evidence for assertions of the form '$\Gamma \vdash \phi$ is not valid.'

Semantics, on the other hand, works in the opposite way. To show that ψ is *not* a consequence of Γ is the 'easy' bit: find a model in which all ϕ_i are true, but ψ isn't. Showing that ψ is a consequence of Γ, on the other hand, is harder in principle. For propositional logic, you need to show that every valuation (an assignment of truth values to all atoms involved) that makes all ϕ_i true also makes ψ true. If there is a small number of valuations, this is not so bad. However, when we look at predicate logic, we will find that there are infinitely many valuations, called *models* from hereon, to consider. Thus, in semantics we have a 'negative' characterisation of the logic. We find establishing assertions of the form '$\Gamma \nvDash \psi$' (ψ is not a semantic entailment of all formulas in Γ) easier than establishing '$\Gamma \vDash \psi$' (ψ is a semantic entailment of Γ), for in the former case we need only talk about one model, whereas in the latter we potentially have to talk about infinitely many.

All this goes to show that it is important to study *both* proof theory *and* semantics. For example, if you are trying to show that ψ is not a consequence of Γ and you have a hard time doing that, you might want to change your strategy for a while by trying to prove the validity of $\Gamma \vdash \psi$. If you find a proof, you know for sure that ψ is a consequence of Γ. If you can't find a proof, then your attempts at proving it often provide insights which lead you to the construction of a counter example. The fact that proof theory and semantics for predicate logic are equivalent is amazing, but it does not stop them having separate roles in logic, each meriting close study.

2.4.1 Models

Recall how we evaluated formulas in propositional logic. For example, the formula $(p \lor \neg q) \to (q \to p)$ is evaluated by computing a truth value (T or F) for it, based on a given valuation (assumed truth values for p and q). This activity is essentially the construction of one line in the truth table of $(p \lor \neg q) \to (q \to p)$. How can we evaluate formulas in predicate logic, e.g.

$$\forall x\, \exists y\, ((P(x) \lor \neg Q(y)) \to (Q(x) \to P(y)))$$

which 'enriches' the formula of propositional logic above? Could we simply assume truth values for $P(x)$, $Q(y)$, $Q(x)$ and $P(y)$ and compute a truth value as before? Not quite, since we have to reflect the meaning of the quantifiers $\forall x$ and $\exists y$, their *dependences* and the actual parameters of P and Q – a formula $\forall x\, \exists y\, R(x, y)$ generally means something else other than $\exists y\, \forall x\, R(x, y)$; why? The problem is that variables are place holders for any, or some, unspecified concrete values. Such values can be of almost any kind: students, birds, numbers, data structures, programs and so on.

Thus, if we encounter a formula $\exists y\, \psi$, we try to find some instance of y (some concrete value) such that ψ holds for that particular instance of y. If this succeeds (i.e. there is such a value of y for which ψ holds), then $\exists y\, \psi$ evaluates to T; otherwise (i.e. there is *no* concrete value of y which realises ψ) it returns F. Dually, evaluating $\forall x\, \psi$ amounts to showing that ψ evaluates to T for *all* possible values of x; if this is successful, we know that $\forall x\, \psi$ evaluates to T; otherwise (i.e. there is *some* value of x such that ψ computes F) it returns F. Of course, such evaluations of formulas require a fixed universe of concrete values, the things we are, so to speak, talking about. Thus, the truth value of a formula in predicate logic depends on, and varies with, the actual choice of values and the meaning of the predicate and function symbols involved.

If variables can take on only finitely many values, we can write a program that evaluates formulas in a compositional way. If the root node of ϕ is $\wedge$, $\vee$, $\rightarrow$ or $\neg$, we can compute the truth value of ϕ by using the truth table of the respective logical connective and by computing the truth values of the subtree(s) of that root, as discussed in Chapter 1. If the root is a quantifier, we have sketched above how to proceed. This leaves us with the case of the root node being a predicate symbol P (in propositional logic this was an atom and we were done already). Such a predicate requires n arguments which have to be terms $t_1, t_2, \dots, t_n$. Therefore, we need to be able to assign truth values to formulas of the form $P(t_1, t_2, \dots, t_n)$.

For formulas $P(t_1, t_2, \dots, t_n)$, there is more going on than in the case of propositional logic. For $n = 2$, the predicate P could stand for something like 'the number computed by t_1 is less than, or equal to, the number computed by t_2.' Therefore, we cannot just assign truth values to P directly without knowing the meaning of terms. We require a *model* of all function and predicate symbols involved. For example, terms could denote *real numbers* and P could denote the relation 'less than or equal to' on the set of real numbers.

Definition 2.14 Let $\mathcal{F}$ be a set of function symbols and $\mathcal{P}$ a set of predicate symbols, each symbol with a fixed number of required arguments. A model $\mathcal{M}$ of the pair $(\mathcal{F}, \mathcal{P})$ consists of the following set of data:

1. A non-empty set A, the universe of concrete values;
2. for each nullary function symbol $f \in \mathcal{F}$, a concrete element $f^{\mathcal{M}}$ of A
3. for each $f \in \mathcal{F}$ with arity $n > 0$, a concrete function $f^{\mathcal{M}} \colon A^n \to A$ from A^n, the set of n-tuples over A, to A; and
4. for each $P \in \mathcal{P}$ with arity $n > 0$, a subset $P^{\mathcal{M}} \subseteq A^n$ of n-tuples over A.

The distinction between f and $f^{\mathcal{M}}$ and between P and $P^{\mathcal{M}}$ is most important. The symbols f and P are just that: symbols, whereas $f^{\mathcal{M}}$ and $P^{\mathcal{M}}$ denote a concrete function (or element) and relation in a model $\mathcal{M}$, respectively.

Example 2.15 Let $\mathcal{F} \stackrel{\text{def}}{=} \{i\}$ and $\mathcal{P} \stackrel{\text{def}}{=} \{R, F\}$; where i is a constant, F a predicate symbol with one argument and R a predicate symbol with two arguments. A model $\mathcal{M}$ contains a set of concrete elements A – which may be a set of states of a computer program. The interpretations $i^{\mathcal{M}}$, $R^{\mathcal{M}}$, and $F^{\mathcal{M}}$ may then be a designated initial state, a state transition relation, and a set of final (accepting) states, respectively. For example, let $A \stackrel{\text{def}}{=} \{a, b, c\}$, $i^{\mathcal{M}} \stackrel{\text{def}}{=} a$, $R^{\mathcal{M}} \stackrel{\text{def}}{=} \{(a,a), (a,b), (a,c), (b,c), (c,c)\}$, and $F^{\mathcal{M}} \stackrel{\text{def}}{=} \{b, c\}$. We informally check some formulas of predicate logic for this model:

1. The formula

$$\exists y\, R(i, y)$$

   says that there is a transition from the initial state to some state; this is true in our model, as there are transitions from the initial state a to a, b, and c.
2. The formula

$$\neg F(i)$$

   states that the initial state is not a final, accepting state. This is true in our model as b and c are the only final states and a is the intitial one.
3. The formula

$$\forall x \forall y \forall z\, (R(x, y) \land R(x, z) \to y = z)$$

   makes use of the equality predicate and states that the transition relation is deterministic: all transitions from any state can go to at most one state (there may be no transitions from a state as well). This is false in our model since state a has transitions to b and c.
4. The formula

$$\forall x \exists y\, R(x, y)$$

   states that the model is free of states that deadlock: all states have a transition to some state. This is true in our model: a can move to a, b or c; and b and c can move to c.

Example 2.16 Let $\mathcal{F} \stackrel{\text{def}}{=} \{e, \cdot\}$ and $\mathcal{P} \stackrel{\text{def}}{=} \{\leq\}$, where e is a constant, $\cdot$ is a function of two arguments and $\leq$ is a predicate in need of two arguments as well. Again, we write $\cdot$ and $\leq$ in infix notation as in $(t_1 \cdot t_2) \leq (t \cdot t)$.

The model $\mathcal{M}$ we have in mind has as set A all binary strings, finite words over the alphabet $\{0, 1\}$, including the empty string denoted by ϵ. The interpretation $e^{\mathcal{M}}$ of e is just the empty word ϵ. The interpretation $\cdot^{\mathcal{M}}$ of $\cdot$ is the concatenation of words. For example, $0110 \cdot^{\mathcal{M}} 1110$ equals 01101110. In general, if $a_1a_2 \dots a_k$ and $b_1b_2 \dots b_n$ are such words with $a_i, b_j \in \{0, 1\}$, then $a_1a_2 \dots a_k \cdot^{\mathcal{M}} b_1b_2 \dots b_n$ equals $a_1a_2 \dots a_kb_1b_2 \dots b_n$. Finally, we interpret $\leq$ as the prefix ordering of words. We say that s_1 is a prefix of s_2 if there is a binary word s_3 such that $s_1 \cdot^{\mathcal{M}} s_3$ equals s_2. For example, 011 is a prefix of 011001 and of 011, but 010 is neither. Thus, $\leq^{\mathcal{M}}$ is the set $\{(s_1, s_2) \mid s_1 \text{ is a prefix of } s_2\}$. Here are again some informal model checks:

1. In our model, the formula
$$\forall x\, ((x \leq x \cdot e) \land (x \cdot e \leq x))$$
says that every word is a prefix of itself concatenated with the empty word and conversely. Clearly, this holds in our model, for $s \cdot^{\mathcal{M}} \epsilon$ is just s and every word is a prefix of itself.
2. In our model, the formula
$$\exists y\, \forall x\, (y \leq x)$$
says that there exists a word s that is a prefix of every other word. This is true, for we may chose ϵ as such a word (there is no other choice in this case).
3. In our model, the formula
$$\forall x\, \exists y\, (y \leq x)$$
says that every word has a prefix. This is clearly the case and there are in general multiple choices for y, which are dependent on x.
4. In our model, the formula $\forall x\, \forall y\, \forall z\, ((x \leq y) \to (x \cdot z \leq y \cdot z))$ says that whenever a word s_1 is a prefix of s_2, then s_1s has to be a prefix of s_2s for every word s. This is clearly not the case. For example, take s_1 as 01, s_2 as 011 and s to be 0.
5. In our model, the formula
$$\neg\exists x\, \forall y\, ((x \leq y) \to (y \leq x))$$
says that there is no word s such that whenever s is a prefix of some other word s_1, it is the case that s_1 is a prefix of s as well. This is true since there cannot be such an s. Assume, for the sake of argument, that there were such a word s. Then s is clearly a prefix of $s0$, but $s0$ cannot be a prefix of s since $s0$ contains one more bit than s.

It is crucial to realise that the notion of a model is extremely liberal and open-ended. All it takes is to choose a non-empty set A, whose elements model real-world objects, and a set of concrete functions and relations, one for each function, respectively predicate, symbol. The only mild requirement imposed on all of this is that the concrete functions and relations on A have the same number of arguments as their syntactic counterparts.

However, you, as a designer or implementor of such a model, have the responsibility of choosing your model wisely. Your model should be a sufficiently accurate picture of whatever it is you want to model, but at the same time it should abstract away (= ignore) aspects of the world which are irrelevant from the perspective of your task at hand.

For example, if you build a database of family relationships, then it would be foolish to interpret *father-of*(x, y) by something like 'x is the daughter of y.' By the same token, you probably would not want to have a predicate for 'is taller than,' since your focus in this model is merely on relationships defined by birth. Of course, there are circumstances in which you may want to add additional features to your database.

Given a model $\mathcal{M}$ for a pair $(\mathcal{F}, \mathcal{P})$ of function and predicate symbols, we are now almost in a position to formally compute a truth value for all formulas in predicate logic which involve only function and predicate symbols from $(\mathcal{F}, \mathcal{P})$. There is still one thing, though, that we need to discuss. Given a formula $\forall x\, \phi$ or $\exists x\, \phi$, we intend to check whether ϕ holds for all, respectively some, value a in our model. While this is intuitive, we have no way of expressing this in our syntax: the formula ϕ usually has x as a free variable; $\phi[a/x]$ is well-intended, but ill-formed since $\phi[a/x]$ is *not* a logical formula, for a is not a term but an element of our model.

Therefore we are forced to interpret formulas *relative to an environment.* You may think of environments in a variety of ways. Essentially, they are look-up tables for all variables; such a table l associates with every variable x a value $l(x)$ of the model. So you can also say that environments are functions $l\colon \mathsf{var} \to A$ from the set of variables var to the universe of values A of the underlying model. Given such a look-up table, we can assign truth values to all formulas. However, for some of these computations we need *updated* look-up tables.

Definition 2.17 A look-up table or environment for a universe A of concrete values is a function $l\colon \mathsf{var} \to A$ from the set of variables var to A. For such an l, we denote by $l[x \mapsto a]$ the look-up table which maps x to a and any other variable y to $l(y)$.

Finally, we are able to give a semantics to formulas of predicate logic. For propositional logic, we did this by computing a truth value. Clearly, it suffices to know in which cases this value is $\mathtt{T}$.

Definition 2.18 Given a model $\mathcal{M}$ for a pair $(\mathcal{F}, \mathcal{P})$ and given an environment l, we define the satisfaction relation $\mathcal{M} \vDash_l \phi$ for each logical formula ϕ over the pair $(\mathcal{F}, \mathcal{P})$ and look-up table l by structural induction on ϕ. If $\mathcal{M} \vDash_l \phi$ holds, we say that ϕ computes to $\mathtt{T}$ in the model $\mathcal{M}$ with respect to the environment l.

P: If ϕ is of the form $P(t_1, t_2, \ldots, t_n)$, then we interpret the terms $t_1, t_2, \ldots, t_n$ in our set A by replacing all variables with their values according to l. In this way we compute concrete values $a_1, a_2, \ldots, a_n$ of A for each of these terms, where we interpret any function symbol $f \in \mathcal{F}$ by $f^{\mathcal{M}}$. Now $\mathcal{M} \vDash_l P(t_1, t_2, \ldots, t_n)$ holds iff $(a_1, a_2, \ldots, a_n)$ is in the set $P^{\mathcal{M}}$.

$\forall x$: The relation $\mathcal{M} \vDash_l \forall x\, \psi$ holds iff $\mathcal{M} \vDash_{l[x \mapsto a]} \psi$ holds for all $a \in A$.

$\exists x$: Dually, $\mathcal{M} \vDash_l \exists x\, \psi$ holds iff $\mathcal{M} \vDash_{l[x \mapsto a]} \psi$ holds for some $a \in A$.

$\neg$: The relation $\mathcal{M} \vDash_l \neg\psi$ holds iff it is not the case that $\mathcal{M} \vDash_l \psi$ holds.

$\vee$: The relation $\mathcal{M} \vDash_l \psi_1 \vee \psi_2$ holds iff $\mathcal{M} \vDash_l \psi_1$ or $\mathcal{M} \vDash_l \psi_2$ holds.

$\wedge$: The relation $\mathcal{M} \vDash_l \psi_1 \wedge \psi_2$ holds iff $\mathcal{M} \vDash_l \psi_1$ and $\mathcal{M} \vDash_l \psi_2$ hold.

$\rightarrow$: The relation $\mathcal{M} \vDash_l \psi_1 \rightarrow \psi_2$ holds iff $\mathcal{M} \vDash_l \psi_2$ holds whenever $\mathcal{M} \vDash_l \psi_1$ holds.

We sometimes write $\mathcal{M} \nvDash_l \phi$ to denote that $\mathcal{M} \vDash_l \phi$ does not hold.

There is a straightforward inductive argument on the height of the parse tree of a formula which says that $\mathcal{M} \vDash_l \phi$ holds iff $\mathcal{M} \vDash_{l'} \phi$ holds, whenever l and l' are two environments which are identical on the set of free variables of ϕ. In particular, if ϕ has *no* free variables at all, we then call ϕ a *sentence*; we conclude that $\mathcal{M} \vDash_l \phi$ holds, or does not hold, regardless of the choice of l. Thus, for sentences ϕ we often elide l and write $\mathcal{M} \vDash \phi$ since the choice of an environment l is then irrelevant.

Example 2.19 Let us illustrate the definitions above by means of another simple example. Let $\mathcal{F} \stackrel{\text{def}}{=} \{\mathsf{alma}\}$ and $\mathcal{P} \stackrel{\text{def}}{=} \{\mathsf{loves}\}$ where alma is a constant and loves a predicate with two arguments. The model $\mathcal{M}$ we choose here consists of the privacy-respecting set $A \stackrel{\text{def}}{=} \{a, b, c\}$, the constant function $\mathsf{alma}^{\mathcal{M}} \stackrel{\text{def}}{=} a$ and the predicate $\mathsf{loves}^{\mathcal{M}} \stackrel{\text{def}}{=} \{(a, a), (b, a), (c, a)\}$, which has two arguments as required. We want to check whether the model $\mathcal{M}$ satisfies

None of Alma's lovers' lovers love her.

First, we need to express the, morally worrying, sentence in predicate logic. Here is such an encoding (as we already discussed, different but logically equivalent encodings are possible):

$$\forall x\, \forall y\, (\mathsf{loves}(x, \mathsf{alma}) \wedge \mathsf{loves}(y, x) \rightarrow \neg\mathsf{loves}(y, \mathsf{alma}))\ . \qquad (2.8)$$

Does the model $\mathcal{M}$ satisfy this formula? Well, it does not; for we may choose a for x and b for y. Since (a, a) is in the set $\mathsf{loves}^{\mathcal{M}}$ and (b, a) is in the set $\mathsf{loves}^{\mathcal{M}}$, we would need that the latter does not hold since it is the interpretation of $\mathsf{loves}(y, \mathsf{alma})$; this cannot be.

And what changes if we modify $\mathcal{M}$ to $\mathcal{M}'$, where we keep A and $\mathsf{alma}^{\mathcal{M}}$, but redefine the interpretation of loves as $\mathsf{loves}^{\mathcal{M}'} \stackrel{\text{def}}{=} \{(b, a), (c, b)\}$? Well, now there is exactly one lover of Alma's lovers, namely c; but c is not one of Alma's lovers. Thus, the formula in (2.8) holds in the model $\mathcal{M}'$.

2.4.2 Semantic entailment

In propositional logic, the semantic entailment $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds iff: whenever all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to $\mathtt{T}$, the formula ψ evaluates to $\mathtt{T}$ as well. How can we define such a notion for formulas in predicate logic, considering that $\mathcal{M} \vDash_l \phi$ is indexed with an environment?

Definition 2.20 Let Γ be a (possibly infinite) set of formulas in predicate logic and ψ a formula of predicate logic.

1. Semantic entailment $\Gamma \vDash \psi$ holds iff for all models $\mathcal{M}$ and look-up tables l, whenever $\mathcal{M} \vDash_l \phi$ holds for all $\phi \in \Gamma$, then $\mathcal{M} \vDash_l \psi$ holds as well.
2. Formula ψ is satisfiable iff there is some model $\mathcal{M}$ and some environment l such that $\mathcal{M} \vDash_l \psi$ holds.
3. Formula ψ is valid iff $\mathcal{M} \vDash_l \psi$ holds for all models $\mathcal{M}$ and environments l in which we can check ψ.
4. The set Γ is consistent or satisfiable iff there is a model $\mathcal{M}$ and a look-up table l such that $\mathcal{M} \vDash_l \phi$ holds for all $\phi \in \Gamma$.

In predicate logic, the symbol $\vDash$ is overloaded: it denotes model checks '$\mathcal{M} \vDash \phi$' and semantic entailment '$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$.' Computationally, each of these notions means trouble. First, establishing $\mathcal{M} \vDash \phi$ will cause problems, if done on a machine, as soon as the universe of values A of $\mathcal{M}$ is infinite. In that case, checking the sentence $\forall x\, \psi$, where x is free in ψ, amounts to verifying $\mathcal{M} \vDash_{[x \mapsto a]} \psi$ for infinitely many elements a.

Second, and much more seriously, in trying to verify that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, we have to check things out for *all possible models*, all models which are equipped with the right structure (i.e. they have functions and predicates with the matching number of arguments). This task is impossible to perform mechanically. This should be contrasted to the situation in propositional logic, where the computation of the truth tables for the propositions involved was the basis for computing this relationship successfully.

However, we can sometimes reason that certain semantic entailments are valid. We do this by providing an argument that does not depend on the actual model at hand. Of course, this works only for a very limited number of cases. The most prominent ones are the *quantifier equivalences* which we already encountered in the section on natural deduction. Let us look at a couple of examples of semantic entailment.

Example 2.21 The justification of the semantic entailment

$$\forall x\,(P(x) \to Q(x)) \vDash \forall x\,P(x) \to \forall x\,Q(x)$$

is as follows. Let $\mathcal{M}$ be a model satisfying $\forall x\,(P(x) \to Q(x))$. We need to show that $\mathcal{M}$ satisfies $\forall x\,P(x) \to \forall x\,Q(x)$ as well. On inspecting the definition of $\mathcal{M} \vDash \psi_1 \to \psi_2$, we see that we are done if not every element of our model satisfies P. Otherwise, every element does satisfy P. But since $\mathcal{M}$ satisfies $\forall x\,(P(x) \to Q(x))$, the latter fact forces every element of our model to satisfy Q as well. By combining these two cases (i.e. either all elements of $\mathcal{M}$ satisfy P, or not) we have shown that $\mathcal{M}$ satisfies $\forall x\,P(x) \to \forall x\,Q(x)$.

What about the converse of the above? Is

$$\forall x\,P(x) \to \forall x\,Q(x) \vDash \forall x\,(P(x) \to Q(x))$$

valid as well? Hardly! Suppose that $\mathcal{M}'$ is a model satisfying $\forall x\,P(x) \to \forall x\,Q(x)$. If A' is its underlying set and $P^{\mathcal{M}'}$ and $Q^{\mathcal{M}'}$ are the corresponding interpretations of P and Q, then $\mathcal{M}' \vDash \forall x\,P(x) \to \forall x\,Q(x)$ simply says that, if $P^{\mathcal{M}'}$ equals A', then $Q^{\mathcal{M}'}$ must equal A' as well. However, if $P^{\mathcal{M}'}$ does not equal A', then this implication is vacuously true (remember that $\mathtt{F} \to \cdot = \mathtt{T}$ no matter what $\cdot$ actually is). In this case we do not get any additional constraints on our model $\mathcal{M}'$. After these observations, it is now easy to construct a counter-example model. Let $A' \stackrel{\text{def}}{=} \{a, b\}$, $P^{\mathcal{M}'} \stackrel{\text{def}}{=} \{a\}$ and $Q^{\mathcal{M}'} \stackrel{\text{def}}{=} \{b\}$. Then $\mathcal{M}' \vDash \forall x\,P(x) \to \forall x\,Q(x)$ holds, but $\mathcal{M}' \vDash \forall x\,(P(x) \to Q(x))$ does not.

2.4.3 The semantics of equality

We have already pointed out the open-ended nature of the semantics of predicate logic. Given a predicate logic over a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$, we need only a non-empty set A equipped with concrete functions or elements $f^{\mathcal{M}}$ (for $f \in \mathcal{F}$) and concrete predicates $P^{\mathcal{M}}$ (for $P \in \mathcal{P}$) in A which have the right arities agreed upon in our specification. Of course, we also stressed that most models have natural interpretations of functions and predicates, but central notions like that of semantic entailment ($\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$) really depend on *all possible models*, even the ones that don't seem to make any sense.

Apparently there is no way out of this peculiarity. For example, where would you draw the line between a model that makes sense and one that doesn't? And would any such choice, or set of criteria, not be *subjective*? Such constraints could also forbid a modification of your model if this alteration were caused by a slight adjustment of the problem domain you intended to model. You see that there are a lot of good reasons for maintaining such a liberal stance towards the notion of models in predicate logic.

However, there is one famous exception. Often one presents predicate logic such that there is always a special predicate $=$ available to denote equality (recall Section 2.3.1); it has two arguments and $t_1 = t_2$ has the intended meaning that the terms t_1 and t_2 compute the same thing. We discussed its proof rule in natural deduction already in Section 2.3.1.

Semantically, one recognises the special role of equality by imposing on an interpretation function $=^{\mathcal{M}}$ to be actual equality on the set A of $\mathcal{M}$. Thus, (a, b) is in the set $=^{\mathcal{M}}$ iff a and b are the same elements in the set A. For example, given $A \stackrel{\text{def}}{=} \{a, b, c\}$, the interpretation $=^{\mathcal{M}}$ of equality is forced to be $\{(a, a), (b, b), (c, c)\}$. Hence the semantics of equality is easy, for it is always modelled *extensionally*.

2.5 Undecidability of predicate logic

We continue our introduction to predicate logic with some negative results. Given a formula ϕ in *propositional logic* we can, at least in principle, determine whether $\vDash \phi$ holds: if ϕ has n propositional atoms, then the truth table of ϕ contains 2^n lines; and $\vDash \phi$ holds if, and only if, the column for ϕ (of length 2^n) contains only `T` entries.

The bad news is that such a mechanical procedure, working for all formulas ϕ, cannot be provided in *predicate logic*. We will give a formal proof of this negative result, though we rely on an informal (yet intuitive) notion of computability.

The problem of determining whether a predicate logic formula is valid is known as a *decision problem*. A solution to a decision problem is a program (written in Java, C, or any other common language) that takes problem instances as input and *always* terminates, producing a correct 'yes' or 'no' output. In the case of the decision problem for predicate logic, the input to the program is an arbitrary formula ϕ of predicate logic and the program is correct if it produces 'yes' whenever the input formula is valid and 'no' whenever it is not. Note that the program which solves a decision problem must terminate for all well-formed input: a program which goes on thinking about it for ever is not allowed. The decision problem at hand is this:

Validity in predicate logic. *Given a logical formula ϕ in predicate logic, does $\vDash \phi$ hold, yes or no?*

We now show that this problem is not solvable; we cannot write a correct C or Java program that works for *all* ϕ. It is important to be clear about exactly what we are stating. Naturally, there are some ϕ which can easily be seen to be valid; and others which can easily be seen to be invalid. However, there are also some ϕ for which it is not easy. Every ϕ can, in principle, be discovered to be valid or not, if you are prepared to work arbitrarily hard at it; but there is no *uniform* mechanical procedure for determining whether ϕ is valid which will work for *all* ϕ.

We prove this by a well-known technique called *problem reduction*. That is, we take some other problem, of which we already know that it is not solvable, and we then show that the solvability of *our* problem entails the solvability of the other one. This is a beautiful application of the proof rules $\neg$i and $\neg$e, since we can then infer that our own problem cannot be solvable as well.

The problem that is known not to be solvable, the *Post correspondence problem*, is interesting in its own right and, upon first reflection, does not seem to have a lot to do with predicate logic.

The Post correspondence problem. *Given a finite sequence of pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ such that all s_i and t_i are binary strings of positive length, is there a sequence of indices $i_1, i_2, \ldots, i_n$ with $n \geq 1$ such that the concatenation of strings $s_{i_1} s_{i_2} \ldots s_{i_n}$ equals $t_{i_1} t_{i_2} \ldots t_{i_n}$?*

Here is an *instance* of the problem which we can solve successfully: the concrete correspondence problem instance C is given by a sequence of three pairs $C \stackrel{\text{def}}{=} ((1, 101), (10, 00), (011, 11))$ so

$$\begin{array}{ccc} s_1 \stackrel{\text{def}}{=} 1 & s_2 \stackrel{\text{def}}{=} 10 & s_3 \stackrel{\text{def}}{=} 011 \\ t_1 \stackrel{\text{def}}{=} 101 & t_2 \stackrel{\text{def}}{=} 00 & t_3 \stackrel{\text{def}}{=} 11. \end{array}$$

A solution to the problem is the sequence of indices $(1, 3, 2, 3)$ since $s_1 s_3 s_2 s_3$ and $t_1 t_3 t_2 t_3$ both equal 101110011. Maybe you think that this problem must surely be solvable; but remember that a computational solution would have to be a program that solves *all* such problem instances. Things get a bit tougher already if we look at this (solvable) problem:

$$\begin{array}{cccc} s_1 \overset{\text{def}}{=} 001 & s_2 \overset{\text{def}}{=} 01 & s_3 \overset{\text{def}}{=} 01 & s_4 \overset{\text{def}}{=} 10 \\ t_1 \overset{\text{def}}{=} 0 & t_2 \overset{\text{def}}{=} 011 & t_3 \overset{\text{def}}{=} 101 & t_4 \overset{\text{def}}{=} 001 \end{array}$$

which you are invited to solve by hand, or by writing a program for this specific instance.

Note that the same number can occur in the sequence of indices, as happened in the first example in which 3 occurs twice. This means that the search space we are dealing with is infinite, which should give us some indication that the problem is unsolvable. However, we do not formally prove it in this book. The proof of the following theorem is due to the mathematician A. Church.

Theorem 2.22 The decision problem of validity in predicate logic is undecidable: no program exists which, given any ϕ, decides whether $\vDash \phi$.

PROOF: As said before, we pretend that validity is decidable for predicate logic and thereby solve the (insoluble) Post correspondence problem. Given a correspondence problem instance C:

$$\begin{array}{l} s_1\ s_2\ \dots\ s_k \\ t_1\ t_2\ \dots\ t_k \end{array}$$

we need to be able to construct, within finite space and time and uniformly so for all instances, some formula ϕ of predicate logic such that $\vDash \phi$ holds iff the correspondence problem instance C above has a solution.

As function symbols, we choose a constant e and two function symbols f_0 and f_1 each of which requires one argument. We think of e as the empty string, or word, and f_0 and f_1 symbolically stand for concatenation with 0, respectively 1. So if $b_1 b_2 \dots b_l$ is a binary string of bits, we can code that up as the term $f_{b_l}(f_{b_{l-1}} \dots (f_{b_2}(f_{b_1}(e))) \dots)$. Note that this coding spells that word *backwards*. To facilitate reading those formulas, we abbreviate terms like $f_{b_l}(f_{b_{l-1}} \dots (f_{b_2}(f_{b_1}(t))) \dots)$ by $f_{b_1 b_2 \dots b_l}(t)$.

We also require a predicate symbol P which expects two arguments. The intended meaning of $P(s, t)$ is that there is some sequence of indices $(i_1, i_2, \dots, i_m)$ such that s is the term representing $s_{i_1} s_{i_2} \dots s_{i_m}$ and t represents $t_{i_1} t_{i_2} \dots t_{i_m}$. Thus, s constructs a string using the same sequence of indices as does t; only s uses the s_i whereas t uses the t_i.

Our sentence ϕ has the coarse structure $\phi_1 \wedge \phi_2 \to \phi_3$ where we set

$$\phi_1 \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} P(f_{s_i}(e), f_{t_i}(e))$$

$$\phi_2 \stackrel{\text{def}}{=} \forall v\, \forall w \left(P(v, w) \to \bigwedge_{i=1}^{k} P(f_{s_i}(v), f_{t_i}(w)) \right)$$

$$\phi_3 \stackrel{\text{def}}{=} \exists z\, P(z, z)\ .$$

Our claim is $\vDash \phi$ holds iff the Post correspondence problem C has a solution.

First, let us assume that $\vDash \phi$ holds. Our strategy is to find a model for ϕ which tells us there is a solution to the correspondence problem C simply by inspecting what it means for ϕ to satisfy that particular model. The universe of concrete values A of that model is the set of all finite, binary strings (including the empty string denoted by ϵ).

The interpretation $e^{\mathcal{M}}$ of the constant e is just that empty string ϵ. The interpretation of f_0 is the unary function $f_0^{\mathcal{M}}$ which appends a 0 to a given string, $f_0^{\mathcal{M}}(s) \stackrel{\text{def}}{=} s0$; similarly, $f_1^{\mathcal{M}}(s) \stackrel{\text{def}}{=} s1$ appends a 1 to a given string. The interpretation of P on $\mathcal{M}$ is just what we expect it to be:

$$P^{\mathcal{M}} \stackrel{\text{def}}{=} \{(s,t) \mid \text{there is a sequence of indices } (i_1, i_2, \dots, i_m) \text{ such that } s \text{ equals } s_{i_1} s_{i_2} \dots s_{i_m} \text{ and } t \text{ equals } t_{i_1} t_{i_2} \dots t_{i_m}\}$$

where s and t are binary strings and the s_i and t_i are the data of the correspondence problem C. A pair of strings (s,t) lies in $P^{\mathcal{M}}$ iff, using the same sequence of indices $(i_1, i_2, \dots, i_m)$, s is built using the corresponding s_i and t is built using the respective t_i.

Since $\vDash \phi$ holds we infer that $\mathcal{M} \vDash \phi$ holds, too. We claim that $\mathcal{M} \vDash \phi_2$ holds as well, which says that whenever the pair (s,t) is in $P^{\mathcal{M}}$, then the pair $(s\, s_i,\, t\, t_i)$ is also in $P^{\mathcal{M}}$ for $i = 1, 2, \dots, k$ (you can verify that is says this by inspecting the definition of $P^{\mathcal{M}}$). Now $(s,t) \in P^{\mathcal{M}}$ implies that there is some sequence $(i_1, i_2, \dots, i_m)$ such that s equals $s_{i_1} s_{i_2} \dots s_{i_m}$ and t equals $t_{i_1} t_{i_2} \dots t_{i_m}$. We simply choose the new sequence $(i_1, i_2, \dots, i_m, i)$ and observe that $s\, s_i$ equals $s_{i_1} s_{i_2} \dots s_{i_m} s_i$ and $t\, t_i$ equals $t_{i_1} t_{i_2} \dots t_{i_m} t_i$ and so $\mathcal{M} \vDash \phi_2$ holds as claimed. (Why does $\mathcal{M} \vDash \phi_1$ hold?)

Since $\mathcal{M} \vDash \phi_1 \wedge \phi_2 \to \phi_3$ and $\mathcal{M} \vDash \phi_1 \wedge \phi_2$ hold, it follows that $\mathcal{M} \vDash \phi_3$ holds as well. By definition of ϕ_3 and $P^{\mathcal{M}}$, this tells us there is a solution to C.

Conversely, let us assume that the Post correspondence problem C has some solution, namely the sequence of indices $(i_1, i_2, \dots, i_n)$. Now we have to show that, if $\mathcal{M}'$ is *any* model having a constant $e^{\mathcal{M}'}$, two unary functions,

$f_0^{\mathcal{M}'}$ and $f_1^{\mathcal{M}'}$, and a binary predicate $P^{\mathcal{M}'}$, then that model has to satisfy ϕ. Notice that the root of the parse tree of ϕ is an implication, so this is the crucial clause for the definition of $\mathcal{M}' \vDash \phi$. By that very definition, we are already done if $\mathcal{M}' \nvDash \phi_1$, or if $\mathcal{M}' \nvDash \phi_2$. The harder part is therefore the one where $\mathcal{M}' \vDash \phi_1 \wedge \phi_2$, for in that case we need to verify $\mathcal{M}' \vDash \phi_3$ as well. The way we proceed here is by *interpreting* finite, binary strings in the domain of values A' of the model $\mathcal{M}'$. This is not unlike the coding of an interpreter for one programming language in another. The interpretation is done by a function interpret which is defined inductively on the data structure of finite, binary strings:

$$\begin{aligned}\mathsf{interpret}(\epsilon) &\stackrel{\text{def}}{=} e^{\mathcal{M}'}\\ \mathsf{interpret}(s0) &\stackrel{\text{def}}{=} f_0^{\mathcal{M}'}(\mathsf{interpret}(s))\\ \mathsf{interpret}(s1) &\stackrel{\text{def}}{=} f_1^{\mathcal{M}'}(\mathsf{interpret}(s))\ .\end{aligned}$$

Note that $\mathsf{interpret}(s)$ is defined inductively on the length of s. This interpretation is, like the coding above, backwards; for example, the string 0100110 gets interpreted as $f_0^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_0^{\mathcal{M}'}(f_0^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_0^{\mathcal{M}'}(e^{\mathcal{M}'})))))))$. Note that $\mathsf{interpret}(b_1 b_2 \dots b_l) = f_{b_l}^{\mathcal{M}'}(f_{b_{l-1}}^{\mathcal{M}'}(\dots(f_{b_1}(e^{\mathcal{M}'})\dots)))$ is just the meaning of $f_s(e)$ in A', where s equals $b_1 b_2 \dots b_l$. Using that and the fact that $\mathcal{M}' \vDash \phi_1$, we conclude that $(\mathsf{interpret}(s_i), \mathsf{interpret}(t_i)) \in P^{\mathcal{M}'}$ for $i = 1, 2, \dots, k$. Similarly, since $\mathcal{M}' \vDash \phi_2$, we know that for all $(s, t) \in P^{\mathcal{M}'}$ we have that $(\mathsf{interpret}(s s_i), \mathsf{interpret}(t t_i)) \in P^{\mathcal{M}'}$ for $i = 1, 2, \dots, k$. Using these two facts, starting with $(s, t) = (s_{i_1}, t_{i_1})$, we repeatedly use the latter observation to obtain

$$(\mathsf{interpret}(s_{i_1} s_{i_2} \dots s_{i_n}), \mathsf{interpret}(t_{i_1} t_{i_2} \dots t_{i_n})) \in P^{\mathcal{M}'}. \qquad (2.9)$$

Since $s_{i_1} s_{i_2} \dots s_{i_n}$ and $t_{i_1} t_{i_2} \dots t_{i_n}$ together form a solution of C, they are equal; and therefore $\mathsf{interpret}(s_{i_1} s_{i_2} \dots s_{i_n})$ and $\mathsf{interpret}(t_{i_1} t_{i_2} \dots t_{i_n})$ are the same elements in A', for interpreting the same thing gets you the same result. Hence (2.9) verifies $\exists z\, P(z, z)$ in $\mathcal{M}'$ and thus $\mathcal{M}' \vDash \phi_3$. □

There are two more negative results which we now get quite easily. Recall that a formula ϕ is *satisfiable* if there is some model $\mathcal{M}$ and some environment l such that $\mathcal{M} \vDash_l \phi$ holds. This property is not to be taken for granted; the formula $\exists x\, (P(x) \wedge \neg P(x))$ is clearly unsatisfiable. More interesting is the observation that ϕ is unsatisfiable if, and only if, $\neg\phi$ is valid, i.e. holds in *all* models. This is an immediate consequence of the definitional clause $\mathcal{M} \vDash_l \neg\phi$ for negation. Since we can't compute validity, it follows that we cannot compute satisfiability either.

The other undecidability result comes from the soundness and completeness of predicate logic which, in special form for sentences, reads as

$$\vDash \phi \;\text{ iff }\; \vdash \phi \tag{2.10}$$

which we do not prove in this text. Since we can't decide validity, we cannot decide *provability* either, on the basis of (2.10). One might reflect on that last negative result a bit. It means bad news if one wants to implement perfect theorem provers which can mechanically produce a proof of a given formula, or refute it. It means good news, though, if we like the thought that machines still need a little bit of human help. Creativity seems to have limits if we leave it to machines alone.

2.6 Expressiveness of predicate logic

Predicate logic is much more expressive than propositional logic, having predicate and function symbols, as well as quantifiers. This expressivess comes at the cost of making validity, satisfiability and provability undecidable. The good news, though, is that checking formulas on models is practical; SQL queries over relational databases or XQueries over XML documents are examples of this in practice.

Software models, design standards, and execution models of hardware or programs often are described in terms of directed graphs. Such models $\mathcal{M}$ are interpretations of a two-argument predicate symbol R over a concrete set A of 'states.'

Example 2.23 Given a set of states $A = \{s_0, s_1, s_2, s_3\}$, let $R^{\mathcal{M}}$ be the set $\{(s_0, s_1), (s_1, s_0), (s_1, s_1), (s_1, s_2), (s_2, s_0), (s_3, s_0), (s_3, s_2)\}$. We may depict this model as a directed graph in Figure 2.5, where an edge (a transition) leads from a node s to a node s' iff $(s, s') \in R^{\mathcal{M}}$. In that case, we often denote this as $s \to s'$.

The validation of many applications requires to show that a 'bad' state cannot be reached from a 'good' state. What 'good' and 'bad' mean will depend on the context. For example, a good state may be one in which an integer expression, say $x * (y - 1)$, evaluates to a value that serves as a safe index into an array a of length 10. A bad state would then be one in which this integer expression evaluates to an unsafe value, say 11, causing an 'out-of-bounds exception.' In its essence, deciding whether from a good state one can reach a bad state is the *reachability* problem in directed graphs.

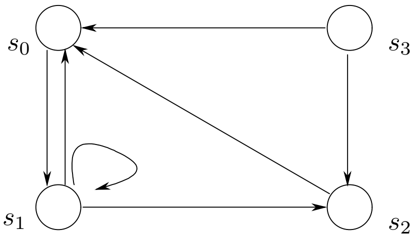


Figure 2.5. A directed graph, which is a model $\mathcal{M}$ for a predicate symbol R with two arguments. A pair of nodes (n, n') is in the interpretation $R^{\mathcal{M}}$ of R iff there is a transition (an edge) from node n to node n' in that graph.

Reachability: Given nodes n and n' in a directed graph, is there a finite path of transitions from n to n'?

In Figure 2.5, state s_2 is reachable from state s_0, e.g. through the path $s_0 \to s_1 \to s_2$. By convention, every state reaches itself by a path of length 0. State s_3, however, is not reachable from s_0; only states s_0, s_1, and s_2 are reachable from s_0. Given the evident importance of this concept, can we express reachability in predicate logic – which is, after all, so expressive that it is undecidable? To put this question more precisely: can we find a predicate-logic formula ϕ with u and v as its only free variables and R as its only predicate symbol (of arity 2) such that ϕ holds in directed graphs iff there is a path in that graph from the node associated to u to the node associated to v? For example, we might try to write:

$$u = v \lor \exists x(R(u,x) \land R(x,v)) \lor \exists x_1 \exists x_2 (R(u,x_1) \land R(x_1,x_2) \land R(x_2,v)) \lor \dots$$

This is infinite, so it's not a well-formed formula. The question is: can we find a well-formed formula with the same meaning?

Surprisingly, this is not the case. To show this we need to record an important consequence of the completeness of natural deduction for predicate logic.

Theorem 2.24 (Compactness Theorem) Let Γ be a set of sentences of predicate logic. If all finite subsets of Γ are satisfiable, then so is Γ.

PROOF: We use proof by contradiction: Assume that Γ is not satisfiable. Then the semantic entailment $\Gamma \vDash \bot$ holds as there is no model in which all $\phi \in \Gamma$ are true. By completeness, this means that the sequent $\Gamma \vdash \bot$ is valid. (Note that this uses a slightly more general notion of sequent in which we may have infinitely many premises at our disposal. Soundness and completeness remain true for that reading.) Thus, this sequent has a proof in natural deduction; this proof – being a finite piece of text – can use only finitely many premises Δ from Γ. But then $\Delta \vdash \bot$ is valid, too, and so $\Delta \vDash \bot$ follows by soundness. But the latter contradicts the fact that all finite subsets of Γ are consistent. □

From this theorem one may derive a number of useful techniques. We mention a technique for ensuring the existence of models of infinite size.

Theorem 2.25 (Löwenheim-Skolem Theorem) Let ψ be a sentence of predicate logic such for any natural number $n \geq 1$ there is a model of ψ with at least n elements. Then ψ has a model with infinitely many elements.

PROOF: The formula $\phi_n \overset{\text{def}}{=} \exists x_1 \exists x_2 \dots \exists x_n \bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)$ specifies that there are at least n elements. Consider the set of sentences $\Gamma \overset{\text{def}}{=} \{\psi\} \cup \{\phi_n \mid n \geq 1\}$ and let Δ be any if its finite subsets. Let $k \geq 1$ be such that $n \leq k$ for all n with $\phi_n \in \Delta$. Since the latter set is finite, such a k has to exist. By assumption, $\{\psi, \phi_k\}$ is satisfiable; but $\phi_k \to \phi_n$ is valid for all $n \leq k$ (why?). Therefore, Δ is satisfiable as well. The compactness theorem then implies that Γ is satisfiable by some model $\mathcal{M}$; in particular, $\mathcal{M} \vDash \psi$ holds. Since $\mathcal{M}$ satisfies ϕ_n for all $n \geq 1$, it cannot have finitely many elements. □

We can now show that reachability is not expressible in predicate logic.

Theorem 2.26 Reachability is not expressible in predicate logic: there is no predicate-logic formula ϕ with u and v as its only free variables and R as its only predicate symbol (of arity 2) such that ϕ holds in directed graphs iff there is a path in that graph from the node associated to u to the node associated to v.

PROOF: Suppose there is a formula ϕ expressing the existence of a path from the node associated to u to the node associated to v. Let c and c' be constants. Let ϕ_n be the formula expressing that there is a path of length n from c to c': we define ϕ_0 as $c = c'$, ϕ_1 as $R(c, c')$ and, for $n > 1$,

$$\phi_n \overset{\text{def}}{=} \exists x_1 \dots \exists x_{n-1}(R(c, x_1) \wedge R(x_1, x_2) \wedge \dots \wedge R(x_{n-1}, c')).$$

Let $\Delta = \{\neg\phi_i \mid i \geq 0\} \cup \{\phi[c/u][c'/v]\}$. All formulas in Δ are sentences and Δ is unsatisfiable, since the 'conjunction' of all sentences in Δ says that there is no path of length 0, no path of length 1, etc. from the node denoted by c to the node denoted by c', but there is a finite path from c to c' as $\phi[c/u][c'/v]$ is true.

However, every finite subset of Δ is satisfiable since there are paths of any finite length. Therefore, by the Compactness Theorem, Δ itself is satisfiable. This is a contradiction. Therefore, there cannot be such a formula ϕ. □

2.6.1 Existential second-order logic

If predicate logic cannot express reachability in graphs, then what can, and at what cost? We seek an extension of predicate logic that can specify such important properties, rather than inventing an entirely new syntax, semantics and proof theory from scratch. This can be realized by applying quantifiers not only to variables, but also to predicate symbols. For a predicate symbol P with $n \geq 1$ arguments, consider formulas of the form

$$\exists P\, \phi \tag{2.11}$$

where ϕ is a formula of predicate logic in which P occurs. Formulas of that form are the ones of *existential second-order logic*. An example of arity 2 is

$$\exists P\, \forall x \forall y \forall z\, (C_1 \wedge C_2 \wedge C_3 \wedge C_4) \tag{2.12}$$

where each C_i is a Horn clause[4]

$$\begin{aligned}
C_1 &\stackrel{\text{def}}{=} P(x, x)\\
C_2 &\stackrel{\text{def}}{=} P(x, y) \wedge P(y, z) \to P(x, z)\\
C_3 &\stackrel{\text{def}}{=} P(u, v) \to \bot\\
C_4 &\stackrel{\text{def}}{=} R(x, y) \to P(x, y).
\end{aligned}$$

If we think of R and P as *two* transition relations on a set of states, then C_4 says that any R-edge is also a P-edge, C_1 states that P is reflexive, C_2 specifies that P is transitive, and C_3 ensures that there is no P-path from the node associated to u to the node associated to v.

Given a model $\mathcal{M}$ with interpretations for all function and predicate symbols of ϕ in (2.11), *except* P, let $\mathcal{M}_T$ be that same model augmented with an interpretation $T \subseteq A \times A$ of P, i.e. $P^{\mathcal{M}_T} = T$. For any look-up table l, the semantics of $\exists P\, \phi$ is then

$$\mathcal{M} \vDash_l \exists P\, \phi \qquad \text{iff} \qquad \text{for some } T \subseteq A \times A,\ \mathcal{M}_T \vDash_l \phi. \tag{2.13}$$

[4] Meaning, a Horn clause after all atomic subformulas are replaced with propositional atoms.

Example 2.27 Let $\exists P\,\phi$ be the formula in (2.12) and consider the model $\mathcal{M}$ of Example 2.23 and Figure 2.5. Let l be a look-up table with $l(u) = s_0$ and $l(v) = s_3$. Does $\mathcal{M} \vDash_l \exists P\,\phi$ hold? For that, we need an interpretation $T \subseteq A \times A$ of P such that $\mathcal{M}_T \vDash_l \forall x \forall y \forall x\, (C_1 \wedge C_2 \wedge C_3 \wedge C_4)$ holds. That is, we need to find a reflexive and transitive relation $T \subseteq A \times A$ that contains $R^{\mathcal{M}}$ but not (s_0, s_3). Please verify that $T \stackrel{\text{def}}{=} \{(s, s') \in A \times A \mid s' \neq s_3\} \cup \{(s_3, s_3)\}$ is such a T. Therefore, $\mathcal{M} \vDash_l \exists P\,\phi$ holds.

In the exercises you are asked to show that the formula in (2.12) holds in a directed graph iff there isn't a finite path from node $l(u)$ to node $l(v)$ in that graph. Therefore, this formula specifies *un*reachability.

2.6.2 Universal second-order logic

Of course, we can negate (2.12) and obtain

$$\forall P\, \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4) \tag{2.14}$$

by relying on the familiar de Morgan laws. This is a formula of *universal second-order logic*. This formula expresses reachability.

Theorem 2.28 Let $\mathcal{M} = (A, R^{\mathcal{M}})$ be any model. Then the formula in (2.14) holds under look-up table l in $\mathcal{M}$ iff $l(v)$ is R-reachable from $l(u)$ in $\mathcal{M}$.

PROOF:

1. First, assume that $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds for all interpretations T of P. Then it also holds for the interpretation which is the reflexive, transitive closure of $R^{\mathcal{M}}$. But for that T, $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ can hold only if $\mathcal{M}_T \vDash_l \neg C_3$ holds, as all other clauses C_i $(i \neq 3)$ are false. But this means that $\mathcal{M}_T \vDash_l P(u, v)$ has to hold. So $(l(u), l(v)) \in T$ follows, meaning that there is a finite path from $l(u)$ to $l(v)$.
2. Conversely, let $l(v)$ be R-reachable from $l(u)$ in $\mathcal{M}$.
   - For any interpretation T of P which is not reflexive, not transitive or does not contain $R^{\mathcal{M}}$ the relation $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds, since T makes one of the clauses $\neg C_1$, $\neg C_2$ or $\neg C_4$ true.
   - The other possibility is that T be a reflexive, transitive relation containing $R^{\mathcal{M}}$. Then T contains the reflexive, transitive closure of $R^{\mathcal{M}}$. But $(l(u), l(v))$ is in that closure by assumption. Therefore, $\neg C_3$ is made true in the interpretation T under look-up table l, and so $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds.

In summary, $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\, (\neg C_1 \lor \neg C_2 \lor \neg C_3 \lor \neg C_4)$ holds for all interpretations $T \subseteq A \times A$. Therefore, $\mathcal{M} \vDash_l \forall P\, \exists x \exists y \exists z\, (\neg C_1 \lor \neg C_2 \lor \neg C_3 \lor \neg C_4)$ holds.

□

It is beyond the scope of this text to show that reachability can also be expressed in *existential* second-order logic, but this is indeed the case. It is an important open problem to determine whether existential second-order logic is closed under negation, i.e. whether for all such formulas $\exists P\, \phi$ there is a formula $\exists Q\, \psi$ of existential second-order logic such that the latter is semantically equivalent to the negation of the former.

If we allow existential and universal quantifiers to apply to predicate symbols in the *same* formula, we arrive at fully-fledged second-order logic, e.g.

$$\exists P \forall Q\, (\forall x \forall y\, (Q(x, y) \to Q(y, x)) \to \forall u \forall v\, (Q(u, v) \to P(u, v))). \tag{2.15}$$

We have $\exists P \forall Q\, (\forall x \forall y\, (Q(x, y) \to Q(y, x)) \to \forall u \forall v\, (Q(u, v) \to P(u, v)))$ iff there is some T such that for all U we have $(\mathcal{M}_T)_U \vDash \forall x \forall y\, (Q(x, y) \to Q(y, x)) \to \forall u \forall v\, (Q(u, v) \to P(u, v))$, the latter being a model check in first-order logic.

If one wants to quantify over relations of relations, one gets third-order logic etc. Higher-order logics require great care in their design. Typical results such as completeness and compactness may quickly fail to hold. Even worse, a naive higher-order logic may be inconsistent at the meta-level. Related problems were discovered in naive set theory, e.g. in the attempt to define the 'set' A that contains as elements those sets X that do not contain themselves as an element:

$$A \stackrel{\text{def}}{=} \{X \mid X \notin X\}. \tag{2.16}$$

We won't study higher-order logics in this text, but remark that many theorem provers or deductive frameworks rely on higher-order logical frameworks.

2.7 Micromodels of software

Two of the central concepts developed so far are

- *model checking*: given a formula ϕ of predicate logic and a matching model $\mathcal{M}$ determine whether $\mathcal{M} \vDash \phi$ holds; and
- *semantic entailment*: given a set of formulas Γ of predicate logic, is $\Gamma \vDash \phi$ valid?

How can we put these concepts to use in the modelling and reasoning about software? In the case of semantic entailment, Γ should contain all the requirements we impose on a software design and ϕ may be a property we think should hold in *any* implementation that meets the requirements Γ. Semantic entailment therefore matches well with software specification and validation; alas, it is undecidable in general. Since model checking is decidable, why not put all the requirements into a model $\mathcal{M}$ and then check $\mathcal{M} \vDash \phi$? The difficulty with this approach is that, by comitting to a particular model $\mathcal{M}$, we are comitting to a lot of detail which doesn't form part of the requirements. Typically, the model instantiates a number of parameters which were left free in the requirements. From this point of view, semantic entailment is better, because it allows a variety of models with a variety of different values for those parameters.

We seek to combine semantic entailment and model checking in a way which attempts to give us the advantages of both. We will extract from the requirements a relatively small number of small models, and check that they satisfy the property ϕ to be proved. This satisfaction checking has the tractability of model checking, while the fact that we range over a set of models (albeit a small one) allows us to consider different values of parameters which are not set in the requirements.

This approach is implemented in a tool called Alloy, due to D. Jackson. The models we consider are what he calls *'micromodels'* of software.

2.7.1 State machines

We illlustrate this approach by revisiting Example 2.15 from page 125. Its models are *state machines* with $\mathcal{F} = \{i\}$ and $\mathcal{P} = \{R, F\}$, where i is a constant, F a predicate symbol with one argument and R a predicate symbol with two arguments. A (concrete) model $\mathcal{M}$ contains a set of concrete elements A – which may be a set of states of a computer program. The interpretations $i^{\mathcal{M}} \in A$, $R^{\mathcal{M}} \in A \times A$, and $F^{\mathcal{M}} \subseteq A$ are understood to be a designated initial state, a state transition relation, and a set of final (accepting) states, respectively. Model $\mathcal{M}$ is concrete since there is nothing left un-specified and all checks $\mathcal{M} \vDash \phi$ have definite answers: they either hold or they don't.

In practice not all functional or other requirements of a software system are known in advance, and they are likely to change during its lifecycle. For example, we may not know how many states there will be; and some transitions may be mandatory whereas others may be optional in an implementation. Conceptually, we seek a description $\mathbb{M}$ of all *compliant* implementations M_i ($i \in I$) of some software system. Given some matching property ψ, we then want to know

- *(assertion checking)* whether ψ holds in all implementations $M_i \in \mathbb{M}$; or
- *(consistency checking)* whether ψ holds in some implementation $M_i \in \mathbb{M}$.

For example, let $\mathbb{M}$ be the set of all concrete models of state machines, as above. A possible assertion check ψ is 'Final states are never initial states.' An example of a consistency check ψ is 'There are state machines that contain a non-final but deadlocked state.'

As remarked earlier, if $\mathbb{M}$ were the set of all state machines, then checking properties would risk being undecidable, and would at least be intractable. If $\mathbb{M}$ consists of a single model, then checking properties would be decidable; but a single model is not general enough. It would comit us to instantiating several parameters which are not part of the requirements of a state machine, such as its size and detailed construction. A better idea is to fix a finite bound on the size of models, and check whether all models of that size that satisfy the requirements also satisfy the property under consideration.

- If we get a positive answer, we are somewhat confident that the property holds in all models. In this case, the answer is not conclusive, because there could be a larger model which fails the property, but nevertheless a positive answer gives us some confidence.
- If we get a negative answer, then we have found a model in $\mathbb{M}$ which violates the property. In that case, we have a conclusive answer, and can inspect the model in question.

D. Jackson's *small scope hypothesis* states that negative answers tend to occur in small models already, boosting the confidence we may have in a positive answer. Here is how one could write the requirements for $\mathbb{M}$ for state machines in Alloy:

```
sig State {}

sig StateMachine {
  A : set State,
  i : A,
  F : set A,
  R : A -> A
}
```

The model specifies two *signatures*. Signature `State` is simple in that it has no internal structure, denoted by {}. Although the states of real systems may well have internal structure, our Alloy declaration abstracts it away. The second signature StateMachine has internal, composite structure, saying that every state machine has a set of states A, an initial state i from A, a set of final states F from A, and a transition relation R of type A -> A. If we read -> as the cartesian product ×, we see that this internal structure is simply the structural information needed for models of Example 2.15 (page 125). Concrete models of state machines are *instances* of signature StateMachine. It is useful to think of signatures as sets whose elements are the instances of that signature. Elements possess all the structure declared in their signature.

Given these signatures, we can code and check an assertion:

```
assert FinalNotInitial {
  all M : StateMachine | no M.i & M.F
} check FinalNotIntial for 3 but 1 StateMachine
```

declares an assertion named FinalNotInitial whose body specifies that for all models M of type StateMachine the property no M.i & M.F is true. Read & for set intersection and no S ('there is no S') for 'set S is empty.' Alloy identifies elements a with singleton sets $\{a\}$, so this set intersection is well typed. The relational dot operator . enables access to the internal components of a state machine: M.i is the initial state of M and M.F is its set of final states etc. Therefore, the expression no M.i & M.F states 'No initial state of M is also a final state of M.' Finally, the check directive informs the analyzer of Alloy that it should try to find a counterexample of the assertion FinalNotInitial with at most three elements for every signature, except for StateMachine which should have at most one.

The results of Alloy's assertion check are shown in Figure 2.7. This visualization has been customized to decorate initial and final states with respective labels i and F. The transition relation is shown as a labeled graph and there is only one transition (from State_0 back to State_0) in this example. Please verify that this is a counterexample to the claim of the assertion FinalNotInitial within the specified scopes. Alloy's GUI lets you search for additional witnesses (here: counterexamples), if they exist.

Similarly, we can check a property of state machines for consistency with our model. Alloy uses the keyword fun for consistency checks. e.g.

```
fun AGuidedSimulation(M : StateMachine, s : M.A) {
  no s.(M.R)
  not s in M.F
  # M.A = 3
} run AGiudedSimulation for 3 but 1 StateMachine
```

```
module AboutStateMachines

sig State {}        -- simple states

sig StateMachine { -- composite state machines
  A : set State,   -- set of states of a state machine
  i : A,           -- initial state of a state machine
  F : set A,       -- set of final states of a state machine
  R : A -> A       -- transition relation of a state machine
}

-- Claim that final states are never initial: false.
assert FinalNotInitial {
  all M : StateMachine | no M.i & M.F
} check FinalNotInitial for 3 but 1 StateMachine

-- Is there a three-state machine with a non-final deadlock? True.
fun AGuidedSimulation(M : StateMachine, s : M.A) {
  no s.(M.R)
  not s in M.F
  # M.A = 3
} run AGuidedSimulation for 3 but 1 StateMachine
```

Figure 2.6. The complete Alloy module for models of state machines, with one assertion and one consistency check. The lexeme -- enables comments on the same line.

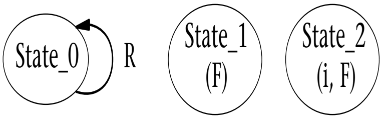


Figure 2.7. Alloy's analyzer finds a state machine model (with one transition only) within the specified scope such that the assertion `FinalNotInitial` is false: the initial state `State_2` is also final.

This consistency check is named `AGuidedSimulation` and followed by an ordered finite list of parameter/type pairs; the first parameter is `M` of type `StateMachine`, the second one is `s` of type `M.A` – i.e. `s` is a state of `M`. The body of a consistency check is a finite list of constraints (here three), which are conjoined implicitly. In this case, we want to find a model with instances of the parameters `M` and `s` such that `s` is a non-final state of `M`, the second constraint `not s in M.F` plus the type information `s : M.A`; and there is no transition out of `s`, the first constraint `no s.(M.R)`.

The latter requires further explanation. The keyword `no` denotes 'there is no;' here it is applied to the set `s.(M.R)`, expressing that there are no

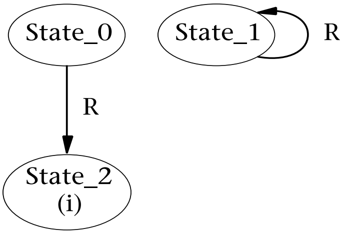


Figure 2.8. Alloy's analyzer finds a state machine model within the specified scope such that the consistency check `AGuidedSimulation` is true: there is a non-final deadlocked state, here `State_2`.

elements in `s.(M.R)`. Since `M.R` is the transition relation of `M`, we need to understand how `s.(M.R)` constructs a set. Well, `s` is an element of `M.A` and `M.R` has type `M.A -> M.A`. Therefore, we may form the set of all elements `s'` such that there is a `M.R`-transition from `s` to `s'`; this is the set `s.(M.R)`. The third constraint states that `M` has exactly three states: in Alloy, `# S = k` declares that the set `S` has exactly k elements.

The **`run`** directive instructs to check the consistency of `AGuidedSimulation` for at most one state machine and at most three states; the constraint analyzer of Alloy returns the witness (here: an example) of Figure 2.8. Please check that this witness satisfies all constraints of the consistency check and that it is within the specified scopes.

The complete model of state machines with these two checks is depicted in Figure 2.6. The keyword plus name `module AboutStateMachines` identify this under-specified model $\mathbb{M}$, rightly suggesting that Alloy is a modular specification and analysis platform.

2.7.2 Alma – re-visited

Recall Example 2.19 from page 128. Its model had three elements and did not satisfy the formula in (2.8). We can now write a module in Alloy which checks whether all *smaller* models have to satisfy (2.8). The code is given in Figure 2.9. It names the module `AboutAlma` and defines a simple signature of type `Person`. Then it declares a signature `SoapOpera` which has a `cast` – a set of type `Person` – a designated cast member `alma`, and a relation `loves` of type `cast -> cast`. We check the assertion `OfLovers` in a scope of at most two persons and at most one soap opera. The body of that assertion is the typed version of (2.8) and deserves a closer look:

1. Expressions of the form `all x : T | F` state that formula `F` is true for all instances `x` of type `T`. So the assertion states that `with S` $\{\ldots\}$ is true for all soap operas `S`.

```
module AboutAlma

sig Person {}

sig SoapOpera {
  cast : set Person,
  alma : cast,
  loves : cast -> cast
}

assert OfLovers {
  all S : SoapOpera |
    with S {
      all x, y : cast |
        alma in x.loves && x in y.loves => not alma in y.loves
    }
}
check OfLovers for 2 but 1 SoapOpera
```

Figure 2.9. In this module, the analysis of OfLovers checks whether there is a model of ≤ 2 persons and ≤ 1 soap operas for which the query in (2.8), page 128, is false.

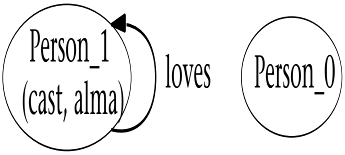


Figure 2.10. Alloy's analyzer finds a counterexample to the formula in (2.8): Alma is the only cast member and loves herself.

2. The expression with S $\{\ldots\}$ is a convenient notation that allows us to write loves and cast instead of the needed S.loves and S.cast (respectively) within its curly brackets.
3. Its body $\ldots$ states that for all x, and y in the cast of S, if alma is loved by x and x is loved by y, then – the symbol => expresses implication – alma is not loved by y.

Alloy's analysis finds a counterexample to this assertion, shown in Figure 2.10. It is a counterexample since alma is her own lover, and therefore also one of her lover's lovers'. Apparently, we have underspecified our model: we implicitly made the domain-specific assumption that self-love makes for

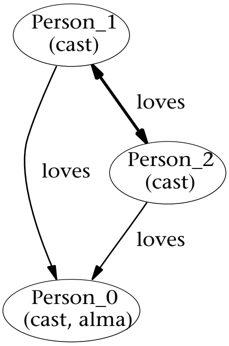


Figure 2.11. Alloy's analyzer finds a counterexample to the formula in (2.8) that meets the constraint of `NoSelfLove` with three cast members. The bidirectional arrow indicates that `Person_1` loves `Person_2` and vice versa.

a poor script of jealousy and intrigue, but *did not rule out* self-love in our Alloy module. To remedy this, we can add a `fact` to the module; facts may have names and restrict the set of possible models: assertions and consistency checks are conducted only over concrete models that satisfy *all* facts of the module. Adding the declaration

```
fact NoSelfLove {
  all S : SoapOpera, p : S.cast | not p in p.(S.loves)
}
```

to the module `AboutAlma` enforces that no member of any soap-opera cast loves him or herself. We re-check the assertion and the analyzer informs us that no solution was found. This suggests that our model from Example 2.19 is indeed a minimal one in the presence of that domain assumption. If we retain that fact, but change the occurrence of `2` in the `check` directive to `3`, we get a counterexample, depicted in Figure 2.11. Can you see why it is a counterexample?

2.7.3 A software micromodel

So far we used Alloy to generate instances of models of first-order logic that satisfy certain constraints expressed as formulas of first-order logic. Now we apply Alloy and its constraint analyzer to a more serious task: we model a software system. The intended benefits provided by a system model are

1. it captures formally static and dynamic system structure and behaviour;
2. it can verify consistency of the constrained design space;

3. it is executable, so it allows guided simulations through a potentially very complex design space; and
4. it can boost our confidence into the correctness of claims about static and dynamic aspects of *all* its compliant implementations.

Moreover, formal models attached to software products can be seen as a *reliability contract;* a promise that the software implements the structure and behaviour of the model and is expected to meet all of the assertions certified therein. (However, this may not be very useful for extremely under-specified models.)

We will model a *software package dependency system.* This system is used when software packages are installed or upgraded. The system checks to see if prerequisites in the form of libraries or other packages are present. The requirements on a software package dependency system are not straightforward. As most computer users know, the upgrading process can go wrong in various ways. For example, upgrading a package can involve replacing shared libraries with newer versions. But other packages which rely on the older versions of the shared libraries may then cease to work.

Software package dependency systems are used in several computer systems, such as Red Hat Linux, .NET's Global Assembly Cache and others. Users often have to guess how technical questions get resolved within the dependency system. To the best of our knowledge, there is no publicly available formal and executable model of any particular dependency system to which application programmers could turn if they had such non-trivial technical questions about its inner workings.

In our model, applications are built out of components. Components offer services to other components. A service can be a number of things. Typically, a service is a method (a modular piece of program code), a field entry, or a type – e.g. the type of a class in an object-oriented programming language. Components typically require the import of services from other components. Technically speaking, such import services resolve all un-resolved references within that component, making the component linkable. A component also has a name and may have a special service, called 'main.'

We model components as a signature in Alloy:

```
sig Component {
  name: Name,            -- name of the component
  main: option Service, -- component may have a 'main' service
  export: set Service,  -- services the component exports
  import: set Service,  -- services the component imports
  version: Number       -- version number of the component
}{ no import & export }
```

The signatures `Service` and `Name` won't require any composite structure for our modelling purposes. The signature `Number` will get an ordering later on. A component is an instance of `Component` and therefore has a `name`, a set of services `export` it offers to other components, and a set `import` of services it needs to import from other components. Last but not least, a component has a `version` number. Observe the role of the modifiers `set` and `option` above.

A declaration `i : set S` means that `i` is a subset of set `S`; but a declaration `i : option S` means that `i` is a subset of `S` with *at most one element*. Thus, `option` enables us to model an element that may (non-empty, singleton set) or may not (empty set) be present; a very useful ability indeed. Finally, a declaration `i : S` states that `i` is a subset of `S` containing *exactly one element*; this really specifies a scalar/element of type `S` since Alloy identifies elements a with sets $\{a\}$.

We can constrain all instances of a signature with `C` by adding { `C` } to its signature declaration. We did this for the signature `Component`, where `C` is the constraint `no import & export`, stating that, in all components, the intersection (`&`) of `import` and `export` is empty (`no`).

A Package Dependency System (PDS) consists of a set of `components`:

```
sig PDS {
  components : set Component
...
}{ components.import in components.export }
```

and other structure that we specify later on. The primary concern in a PDS is that its set of components be *coherent*: at all times, all imports of all of its components can be serviced within that PDS. This requirement is enforced for all instances of `PDS` by adding the constraint `components.import in components.export` to its signature. Here `components` is a *set* of components and Alloy defines the meaning of `components.import` as the *union* of all sets `c.import`, where `c` is an element of `components`. Therefore the requirement states that, for all `c` in `components`, all of `c`'s needed services can be provided by some component in `components` as well. This is exactly the integrity constraint we need for the set of components of a PDS. Observe that this requirement does not specify which component provides which service, which would be an unacceptable imposition on implementation freedom.

Given this integrity constraint we can already model the installation (adding) or removal of a component in a PDS, without having specified the remaining structure of a PDS. This is possible since, in the context of these operations, we may abstract a PDS into its set of `components`. We model the addition of a component to a PDS as a parametrized fun-statement with name AddComponent and three parameters

```
fun AddComponent(P, P’: PDS, c: Component) {
  not c in P.components
  P’.components = P.components + c
} run AddComponent for 3
```

where P is intended to be the PDS *prior* to the execution of that operation, P’ models the PDS *after* that execution, and c models the component that is to be added. This intent interprets the parametric constraint AddComponent as an *operation* leading from one ‘state’ to another (obtained by removing c from the PDS P). The body of AddComponent states two constraints, conjoined implicitly. Thus, this operation applies only if the component c is not already in the set of components of the PDS (not c in P.components; an example of a *precondition*) and if the PDS adds only c and does not lose any other components (P’.components = P.components + c; an example of a *postcondition*).

To get a feel for the complexities and vexations of designing software systems, consider our conscious or implicit decision to enforce that all instances of PDS have a coherent set of components. This sounds like a very good idea, but what if a ‘real’ and faulty PDS ever gets to a state in which it is incoherent? We would then be prevented from adding components that may restore its coherence! Therefore, the aspects of our model do not include issues such as repair – which may indeed by an important software management aspect.

The specification for the removal of a component is very similar to the one for AddComponent:

```
fun RemoveComponent(P, P’: PDS, c: Component) {
  c in P.components
  P’.components = P.components - c
} run RemoveComponent for 3
```

except that the precondition now insists that c be in the set of components of the PDS prior to the removal; and the postcondition specifies that the PDS lost component c but did not add or lose any other components. The expression S - T denotes exactly those ‘elements’ of S that are not in T.

It remains to complete the signature for PDS. Three additions are made.

1. A relation schedule assigns to each PDS component and any of its import services a component in that PDS that provides that service.

```
fact SoundPDSs {
  all P : PDS |
    with P {
    all c : components, s : Service |  --1
      let c’ = c.schedule[s] {
      (some c’ iff s in c.import) && (some c’ => s in c’.export)
     }
    all c : components | c.requires = c.schedule[Service]  --2
    }
}
```

Figure 2.12. A fact that constrains the state and schedulers of all PDSs.

2. Derived from `schedule` we obtain a relation `requires` between components of the PDS that expresses the dependencies between these components based on the `schedule`.
3. Finally, we add constraints that ensure the integrity and correct handling of `schedule` and `requires` for *all instances of* `PDS`.

The complete signature of `PDS` is

```
sig PDS {
  components : set Component,
  schedule   : components -> Service ->? components,
  requires   : components -> components
}
```

For any `P : PDS`, the expression `P.schedule` denotes a relation of type `P.components -> Service ->? P.components`. The `?` is a *multiplicity constraint*, saying that each component of the PDS and each service get related to *at most one* component. This will ensure that the scheduler is deterministic and that it may not schedule anything – e.g. when the service is not needed by the component in the first argument. In Alloy there are also multiplicity markings `!` for ‘exactly one’ and `+` for ‘one or more.’ The absence of such markings means ‘zero or more.’ For example, the declaration of `requires` uses that default reading.

We use a `fact`-statement to constrain even further the structure and behaviour of all PDSs, depicted in Figure 2.12. The fact named `SoundPDSs` quantifies the constraints over all instances of PDSs (`all P : PDS | ...`) and uses `with P` {...} to avoid the use of navigation expressions of the form `P.e`. The body of that fact lists two constraints `--1` and `--2`:

`--1` states two constraints within a `let`-expression of the form `let x = E {...}`. Such a `let`-expression declares all free occurrences of `x` in `{...}` to be equal to `E`. Note that `[]` is a version of the dot operator `.` with lower binding priority, so `c.schedule[s]` is syntactic sugar for `s.(c.schedule)`.

- In the first constraint, component `c` and a service `s` have another component `c'` scheduled (`some c'` is true iff set `c'` is non-empty) if and only if `s` is actually in the import set of `c`. Only needed services are scheduled!
- In the second constraint, if `c'` is scheduled to provide service `s` for `c`, then `s` is in the export set of `c'` – we can only schedule components that can provide the scheduled services!

`--2` defines `requires` in terms of `schedule`: a component `c` requires all those components that are scheduled to provide some service for `c`.

Our complete Alloy model for PDSs is shown in Figure 2.13. Using Alloy's constraint analyzer we validate that all our `fun`-statements, notably the operations of removing and adding components to a PDS, are logically consistent for this design.

The assertion `AddingIsFunctionalForPDSs` claims that the execution of the operation which adds a component to a PDS renders a unique result PDS. Alloy's analyzer finds a counterexample to this claim, where `P` has no components, so nothing is scheduled or required; and `P'` and `P''` have `Component_2` as only component, added to `P`, so this component is required and scheduled in those PDSs.

Since `P'` and `P''` seem to be equal, how can this be a counterexample? Well, we ran the analysis in scope 3, so `PDS = {PDS_0, PDS_1, PDS_2}` and Alloy chose `PDS_0` as `P`, `PDS_1` as `P'`, and `PDS_2` as `P''`. Since the set `PDS` contains three elements, Alloy 'thinks' that they are all different from each other. This is the interpretation of equality enforced by predicate logic. Obviously, what is needed here is a *structural equality of types*: we want to ensure that the addition of a component results into a PDS with unique structure. A `fun`-statement can be used to specify structural equality:

```
fun StructurallyEqual(P, P' : PDS) {
  P.components = P'.components
  P.schedule = P'.schedule
  P.requires = P'.requires
} run StructurallyEqual for 2
```

We then simply replace the expression `P' = P''` in `AdditionIsFunctional` with the expression `StructurallyEqual(P',P'')`, increase the scope for

```
module PDS

open std/ord    -- opens specification template for linear order

sig Component {
  name: Name,
  main: option Service,
  export: set Service,
  import: set Service,
  version: Number
}{ no import & export }

sig PDS {
  components: set Component,
  schedule: components -> Service ->? components,
  requires: components -> components
}{ components.import in components.export }

fact SoundPDSs {
  all P : PDS |
    with P {
    all c : components, s : Service |  --1
      let c’ = c.schedule[s] {
        (some c’ iff s in c.import) && (some c’ => s in c’.export) }
    all c : components | c.requires = c.schedule[Service]  }  --2
}

sig Name, Number, Service {}

fun AddComponent(P, P’: PDS, c: Component) {
 not c in P.components
 P’.components = P.components + c
} run AddComponent for 3 but 2 PDS

fun RemoveComponent(P, P’: PDS, c : Component) {
  c in P.components
  P’.components = P.components - c
} run RemoveComponent for 3 but 2 PDS

fun HighestVersionPolicy(P: PDS) {
  with P {
    all s : Service, c : components, c’ : c.schedule[s],
    c’’ : components - c’ {
       s in c’’.export && c’’.name = c’.name =>
         c’’.version in c’.version.^(Ord[Number].prev) } }
} run HighestVersionPolicy for 3 but 1 PDS

fun AGuidedSimulation(P,P’,P’’ : PDS, c1, c2 : Component) {
  AddComponent(P,P’,c1)    RemoveComponent(P,P’’,c2)
  HighestVersionPolicy(P)  HighestVersionPolicy(P’)  HighestVersionPolicy(P’’)
} run AGuidedSimulation for 3

assert AddingIsFunctionalForPDSs {
  all P, P’, P’’: PDS, c: Component   {
    AddComponent(P,P’,c) &&
    AddComponent(P,P’’,c) => P’ = P’’ }
} check AddingIsFunctionalForPDSs for 3
```

Figure 2.13. Our Alloy model of the PDS.

that assertion to 7, re-built the model, and re-analyze that assertion. Perhaps surprisingly, we find as counterexample a PDS_0 with two components Component_0 and Component_1 such that Component_0.import = { Service_2 } and Component_1.import = { Service_1 }. Since Service_2 is contained in Component_2.export, we have two structurally different legitimate post states which are obtained by adding Component_2 but which differ in their scheduler. In P' we have the same scheduling instances as in PDS_0. Yet P'' schedules Component_2 to provide service Service_2 for Component_0; and Component_0 still provides Service_1 to Component_1. This analysis reveals that the addition of components creates opportunities to reschedule services, for better (e.g. optimizations) or for worse (e.g. security breaches).

The utility of a micromodel of software resides perhaps more in the ability to explore it through guided simulations, as opposed to verifying some of its properties with absolute certainty. We demonstrate this by generating a simulation that shows the removal and the addition of a component to a PDS such that the scheduler always schedules components with the highest version number possible in all PDSs. Therefore we know that such a scheduling policy is consistent for these two operations; it is by no means the only such policy and is not guaranteed to ensure that applications won't break when using scheduled services. The fun-statement

```
fun HighestVersionPolicy(P: PDS) {
  with P {
    all s : Service, c : components, c' : c.schedule[s],
    c'' : components - c' {
       s in c''.export && c''.name = c'.name =>
         c''.version in c'.version.^(Ord[Number].prev)
    }
  }
} run HighestVersionPolicy for 3 but 1 PDS
```

specifies that, among those suppliers with identical name, the scheduler chooses one with the highest available version number. The expression

```
c'.version.^(Ord[Number].prev)
```

needs explaining: c'.version is the version number of c', an element of type Number. The symbol ^ can be applied to a binary relation r : T -> T such that ^r has again type T -> T and denotes the *transitive closure* of r. In this case, T equals Number and r equals Ord[Number].prev.

But what shall me make of the latter expression? It assumes that the module contains a statement `open std/ord` which opens the signature specifications from another module in file `ord.als` of the library `std`. That module contains a signature named `Ord` which has a type variable as a parameter; it is *polymorphic*. The expression `Ord[Number]` instantiates that type variable with the type `Number`, and then invokes the `prev` relation of that signature with that type, where `prev` is constrained in `std/ord` to be a linear order. The net effect is that we create a linear order on `Number` such that `n.prev` is the previous element of `n` with respect to that order. Therefore, `n.^prev` lists all elements that are smaller than `n` in that order. Please reread the body of that `fun`-statement to convince yourself that it states what is intended.

Since `fun`-statements can be invoked with instances of their parameters, we can write the desired simulation based on `HighestVersionPolicy`:

```
fun AGuidedSimulation(P,P',P'' : PDS, c1, c2 : Component) {
  AddComponent(P,P',c1)   RemoveComponent(P,P'',c2)
  HighestVersionPolicy(P)
  HighestVersionPolicy(P') HighestVersionPolicy(P'')
} run AGuidedSimulation for 3
```

Alloy's analyzer generates a scenario for this simulation, which amounts to two different operation snapshots originating in `P` such that all three participating PDSs schedule according to `HighestVersionPolicy`. Can you spot why we had to work with two components `c1` and `c2`?

We conclude this case study by pointing out limitations of Alloy and its analyzer. In order to be able to use a SAT solver for propositional logic as an analysis engine, we can only check or run formulas of existential or universal second-order logic in the bodies of assertions or in the bodies of `fun`-statements (if they are wrapped in existential quantifiers for all parameters). For example, we cannot even check whether there is an instance of `AddComponent` such that for the resulting PDS a certain scheduling policy is *impossible*. For less explicit reasons it also seems unlikely that we can check in Alloy that every coherent set of components is realizable as `P.components` for some PDS `P`. This deficiency is due to the inherent complexity of such problems and theorem provers may have to be used if such properties need to be guaranteed. On the other hand, the expressiveness of Alloy allows for the rapid prototyping of models and the exploration of simulations and possible counterexamples which should enhance once understanding of a design and so improve that design's reliability.

2.8 Exercises

Exercises 2.1

* 1. Use the predicates

$$\begin{aligned} A(x,y)\colon&\quad x \text{ admires } y\\ B(x,y)\colon&\quad x \text{ attended } y\\ P(x)\colon&\quad x \text{ is a professor}\\ S(x)\colon&\quad x \text{ is a student}\\ L(x)\colon&\quad x \text{ is a lecture} \end{aligned}$$

and the nullary function symbol (constant)

$$m\colon\quad \text{Mary}$$

to translate the following into predicate logic:

(a) Mary admires every professor.
(The answer is not $\forall x\, A(m, P(x))$.)
(b) Some professor admires Mary.
(c) Mary admires herself.
(d) No student attended every lecture.
(e) No lecture was attended by every student.
(f) No lecture was attended by any student.

2. Use the predicate specifications

$$\begin{aligned} B(x,y)\colon&\quad x \text{ beats } y\\ F(x)\colon&\quad x \text{ is an (American) football team}\\ Q(x,y)\colon&\quad x \text{ is quarterback of } y\\ L(x,y)\colon&\quad x \text{ loses to } y \end{aligned}$$

and the constant symbols

$$\begin{aligned} c\colon&\quad \text{Wildcats}\\ j\colon&\quad \text{Jayhawks} \end{aligned}$$

to translate the following into predicate logic.

(a) Every football team has a quarterback.
(b) If the Jayhawks beat the Wildcats, then the Jayhawks do not lose to every football team.
(c) The Wildcats beat some team, which beat the Jayhawks.

* 3. Find appropriate predicates and their specification to translate the following into predicate logic:

(a) All red things are in the box.
(b) Only red things are in the box.
(c) No animal is both a cat and a dog.
(d) Every prize was won by a boy.
(e) A boy won every prize.

4. Let $F(x, y)$ mean that x is the father of y; $M(x, y)$ denotes x is the mother of y. Similarly, $H(x, y)$, $S(x, y)$, and $B(x, y)$ say that x is the husband/sister/brother of y, respectively. You may also use constants to denote individuals, like 'Ed' and 'Patsy.' However, you are not allowed to use any predicate symbols other than the above to translate the following sentences into predicate logic:
   (a) Everybody has a mother.
   (b) Everybody has a father and a mother.
   (c) Whoever has a mother has a father.
   (d) Ed is a grandfather.
   (e) All fathers are parents.
   (f) All husbands are spouses.
   (g) No uncle is an aunt.
   (h) All brothers are siblings.
   (i) Nobody's grandmother is anybody's father.
   (j) Ed and Patsy are husband and wife.
   (k) Carl is Monique's brother-in-law.
5. The following sentences are taken from the RFC3157 Internet Taskforce Document 'Securely Available Credentials – Requirements.' Specify each sentence in predicate logic, defining predicate symbols as appropriate:
   (a) An attacker can persuade a server that a successful login has occurred, even if it hasn't.
   (b) An attacker can overwrite someone else's credentials on the server.
   (c) All users enter passwords instead of names.
   (d) Credential transfer both to and from a device MUST be supported.
   (e) Credentials MUST NOT be forced by the protocol to be present in cleartext at any device other than the end user's.
   (f) The protocol MUST support a range of cryptographic algorithms, including syymetric and asymmetric algorithms, hash algorithms, and MAC algorithms.
   (g) Credentials MUST only be downloadable following user authentication or else only downloadable in a format that requires completion of user authentication for deciphering.
   (h) Different end user devices MAY be used to download, upload, or manage the same set of credentials.

Exercises 2.2

1. Let $\mathcal{F}$ be $\{d, f, g\}$, where d is a constant, f a function symbol with two arguments and g a function symbol with three arguments.
   (a) Which of the following strings are terms over $\mathcal{F}$? Draw the parse tree of those strings which are indeed terms:
      i. $g(d, d)$
      * ii. $f(x, g(y, z), d)$

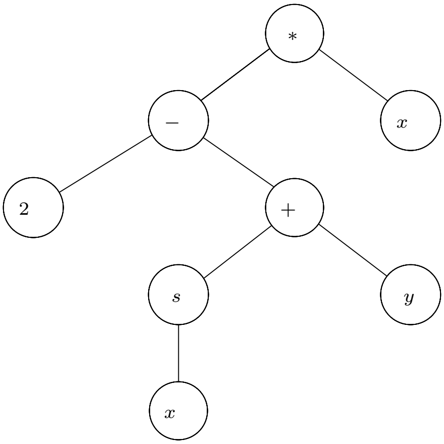


Figure 2.14. A parse tree representing an arithmetic term.

* iii. $g(x, f(y, z), d)$

iv. $g(x, h(y, z), d)$

v. $f(f(g(d, x), f(g(d, x), y, g(y, d)), g(d, d)), g(f(d, d, x), d), z)$

(b) The length of a term over $\mathcal{F}$ is the length of its string representation, where we count all commas and parentheses. For example, the length of $f(x, g(y, z), z)$ is 13. List all variable-free terms over $\mathcal{F}$ of length less than 10.

* (c) The height of a term over $\mathcal{F}$ is defined as 1 plus the length of the longest path in its parse tree, as in Definition 1.32. List all variable-free terms over $\mathcal{F}$ of height less than 4.

2. Draw the parse tree of the term $(2 - s(x)) + (y * x)$, considering that $-$, $+$, and $*$ are used in infix in this term. Compare your solution with the parse tree in Figure 2.14.

3. Which of the following strings are formulas in predicate logic? Specify a reason for failure for strings which aren't, draw parse trees of all strings which are.

* (a) Let m be a constant, f a function symbol with one argument and S and B two predicate symbols, each with two arguments:

i. $S(m, x)$

ii. $B(m, f(m))$

iii. $f(m)$

iv. $B(B(m, x), y)$

v. $S(B(m), z)$

vi. $(B(x, y) \to (\exists z\, S(z, y)))$

vii. $(S(x, y) \to S(y, f(f(x))))$

viii. $(B(x) \to B(B(x)))$.

(b) Let c and d be constants, f a function symbol with one argument, g a function symbol with two arguments and h a function symbol with three arguments. Further, P and Q are predicate symbols with three arguments:

i. $\forall x\, P(f(d), h(g(c,x), d, y))$
ii. $\forall x\, P(f(d), h(P(x,y), d, y))$
iii. $\forall x\, Q(g(h(x, f(d), x), g(x,x)), h(x,x,x), c)$
iv. $\exists z\, (Q(z,z,z) \to P(z))$
v. $\forall x\, \forall y\, (g(x,y) \to P(x,y,x))$
vi. $Q(c,d,c)$.

4. Let ϕ be $\exists x\, (P(y,z) \land (\forall y\, (\neg Q(y,x) \lor P(y,z))))$, where P and Q are predicate symbols with two arguments.
 * (a) Draw the parse tree of ϕ.
 * (b) Identify all bound and free variable leaves in ϕ.
 (c) Is there a variable in ϕ which has free and bound occurrences?
 * (d) Consider the terms w (w is a variable), $f(x)$ and $g(y,z)$, where f and g are function symbols with arity 1 and 2, respectively.
  i. Compute $\phi[w/x]$, $\phi[w/y]$, $\phi[f(x)/y]$ and $\phi[g(y,z)/z]$.
  ii. Which of w, $f(x)$ and $g(y,z)$ are free for x in ϕ?
  iii. Which of w, $f(x)$ and $g(y,z)$ are free for y in ϕ?
 (e) What is the scope of $\exists x$ in ϕ?
 * (f) Suppose that we change ϕ to $\exists x\, (P(y,z) \land (\forall x\, (\neg Q(x,x) \lor P(x,z))))$. What is the scope of $\exists x$ now?
5. (a) Let P be a predicate symbol with arity 3. Draw the parse tree of $\psi \stackrel{\text{def}}{=} \neg(\forall x\, ((\exists y\, P(x,y,z)) \land (\forall z\, P(x,y,z))))$.
 (b) Indicate the free and bound variables in that parse tree.
 (c) List all variables which occur free and bound therein.
 (d) Compute $\psi[t/x]$, $\psi[t/y]$ and $\psi[t/z]$, where $t \stackrel{\text{def}}{=} g(f(g(y,y)), y)$. Is t free for x in ψ; free for y in ψ; free for z in ψ?
6. Rename the variables for ϕ in Example 2.9 (page 106) such that the resulting formula ψ has the same meaning as ϕ, but $f(y,y)$ is free for x in ψ.

Exercises 2.3

1. Prove the validity of the following sequents using, among others, the rules =i and =e. Make sure that you indicate for each application of =e what the rule instances ϕ, t_1 and t_2 are.
 (a) $(y = 0) \land (y = x) \vdash 0 = x$
 (b) $t_1 = t_2 \vdash (t + t_2) = (t + t_1)$
 (c) $(x = 0) \lor ((x + x) > 0) \vdash (y = (x + x)) \to ((y > 0) \lor (y = (0 + x)))$.
2. Recall that we use $=$ to express the equality of elements in our models. Consider the formula $\exists x\, \exists y\, (\neg(x = y) \land (\forall z\, ((z = x) \lor (z = y))))$. Can you say, in plain English, what this formula specifies?
3. Try to write down a sentence of predicate logic which intuitively holds in a model iff the model has (respectively)
 * (a) exactly three distinct elements
 (b) at most three distinct elements
 * (c) only finitely many distinct elements.

What 'limitation' of predicate logic causes problems in finding such a sentence for the last item?

4. (a) Find a (propositional) proof for $\phi \to (q_1 \land q_2) \vdash (\phi \to q_1) \land (\phi \to q_2)$.

   (b) Find a (predicate) proof for $\phi \to \forall x\, Q(x) \vdash \forall x\, (\phi \to Q(x))$, provided that x is not free in ϕ.

   (Hint: whenever you used $\land$ rules in the (propositional) proof of the previous item, use $\forall$ rules in the (predicate) proof.)

   (c) Find a proof for $\forall x\, (P(x) \to Q(x)) \vdash \forall x\, P(x) \to \forall x\, Q(x)$.

   (Hint: try $(p_1 \to q_1) \land (p_2 \to q_2) \vdash p_1 \land p_2 \to q_1 \land q_2$ first.)

5. Find a propositional logic sequent that corresponds to $\exists x\, \neg\phi \vdash \neg\forall x\, \phi$. Prove it.

6. Provide proofs for the following sequents:

   (a) $\forall x\, P(x) \vdash \forall y\, P(y)$; using $\forall x\, P(x)$ as a premise, your proof needs to end with an application of $\forall$i which requires the formula $P(y_0)$.

   (b) $\forall x\, (P(x) \to Q(x)) \vdash (\forall x\, \neg Q(x)) \to (\forall x\, \neg P(x))$

   (c) $\forall x\, (P(x) \to \neg Q(x)) \vdash \neg(\exists x\, (P(x) \land Q(x)))$.

7. The sequents below look a bit tedious, but in proving their validity you make sure that you really understand how to nest the proof rules:

   * (a) $\forall x\, \forall y\, P(x, y) \vdash \forall u\, \forall v\, P(u, v)$

   (b) $\exists x\, \exists y\, F(x, y) \vdash \exists u\, \exists v\, F(u, v)$

   * (c) $\exists x\, \forall y\, P(x, y) \vdash \forall y\, \exists x\, P(x, y)$.

8. In this exercise, whenever you use a proof rule for quantifiers, you should mention how its side condition (if applicable) is satisfied.

   (a) Prove 2(b-h) of Theorem 2.13 from page 117.

   (b) Prove one direction of 1(b) of Theorem 2.13: $\neg\exists x\, \phi \vdash \forall x\, \neg\phi$.

   (c) Prove 3(a) of Theorem 2.13: $(\forall x\, \phi) \land (\forall x\, \psi) \dashv\vdash \forall x\, (\phi \land \psi)$; recall that you have to do two separate proofs.

   (d) Prove both directions of 4(a) of Theorem 2.13: $\forall x\, \forall y\, \phi \dashv\vdash \forall y\, \forall x\, \phi$.

9. Prove the validity of the following sequents in predicate logic, where F, G, P, and Q have arity 1, and S has arity 0 (a 'propositional atom'):

   * (a) $\exists x\, (S \to Q(x)) \vdash S \to \exists x\, Q(x)$

   (b) $S \to \exists x\, Q(x) \vdash \exists x\, (S \to Q(x))$

   (c) $\exists x\, P(x) \to S \vdash \forall x\, (P(x) \to S)$

   * (d) $\forall x\, P(x) \to S \vdash \exists x\, (P(x) \to S)$

   (e) $\forall x\, (P(x) \lor Q(x)) \vdash \forall x\, P(x) \lor \exists x\, Q(x)$

   (f) $\forall x\, \exists y\, (P(x) \lor Q(y)) \vdash \exists y\, \forall x\, (P(x) \lor Q(y))$

   (g) $\forall x\, (\neg P(x) \land Q(x)) \vdash \forall x\, (P(x) \to Q(x))$

   (h) $\forall x\, (P(x) \land Q(x)) \vdash \forall x\, (P(x) \to Q(x))$

   (i) $\exists x\, (\neg P(x) \land \neg Q(x)) \vdash \exists x\, (\neg(P(x) \land Q(x)))$

   (j) $\exists x\, (\neg P(x) \lor Q(x)) \vdash \exists x\, (\neg(P(x) \land \neg Q(x)))$

   * (k) $\forall x\, (P(x) \land Q(x)) \vdash \forall x\, P(x) \land \forall x\, Q(x)$.

   * (l) $\forall x\, P(x) \lor \forall x\, Q(x) \vdash \forall x\, (P(x) \lor Q(x))$.

   *(m) $\exists x\, (P(x) \land Q(x)) \vdash \exists x\, P(x) \land \exists x\, Q(x)$.

   * (n) $\exists x\, F(x) \lor \exists x\, G(x) \vdash \exists x\, (F(x) \lor G(x))$.

   (o) $\forall x\, \forall y\, (S(y) \to F(x)) \vdash \exists y S(y) \to \forall x\, F(x)$.

* (p) $\neg\forall x\,\neg P(x) \vdash \exists x\,P(x)$.

* (q) $\forall x\,\neg P(x) \vdash \neg\exists x\,P(x)$.

* (r) $\neg\exists x\,P(x) \vdash \forall x\,\neg P(x)$.

10. Just like natural deduction proofs for propositional logic, certain things that look easy can be hard to prove for predicate logic. Typically, these involve the $\neg\neg$e rule. The patterns are the same as in propositional logic:

(a) Proving that $p \vee q \vdash \neg(\neg p \wedge \neg q)$ is valid is quite easy. Try it.

(b) Show that $\exists x\,P(x) \vdash \neg\forall x\,\neg P(x)$ is valid.

(c) Proving that $\neg(\neg p \wedge \neg q) \vdash p \vee q$ is valid is hard; you have to try to prove $\neg\neg(p \vee q)$ first and then use the $\neg\neg$e rule. Do it.

(d) Re-express the sequent from the previous item such that p and q are unary predicates and both formulas are universally quantified. Prove its validity.

11. The proofs of the sequents below combine the proof rules for equality and quantifiers. We write $\phi \leftrightarrow \psi$ as an abbreviation for $(\phi \to \psi) \wedge (\psi \to \phi)$. Find proofs for

* (a) $P(b) \vdash \forall x\,(x = b \to P(x))$

(b) $P(b),\ \forall x \forall y\,(P(x) \wedge P(y) \to x = y) \vdash \forall x\,(P(x) \leftrightarrow x = b)$

* (c) $\exists x\,\exists y\,(H(x,y) \vee H(y,x)),\ \neg\exists x\,H(x,x) \vdash \exists x \exists y\,\neg(x = y)$

(d) $\forall x\,(P(x) \leftrightarrow x = b) \vdash P(b) \wedge \forall x \forall y\,(P(x) \wedge P(y) \to x = y)$.

* 12. Prove the validity of $S \to \forall x\,Q(x) \vdash \forall x\,(S \to Q(x))$, where S has arity 0 (a 'propositional atom').

13. By natural deduction, show the validity of

* (a) $\forall x\,P(a,x,x),\ \forall x\,\forall y\,\forall z\,(P(x,y,z) \to P(f(x),y,f(z)))$
$\vdash P(f(a),a,f(a))$

* (b) $\forall x\,P(a,x,x),\ \forall x\,\forall y\,\forall z\,(P(x,y,z) \to P(f(x),y,f(z)))$
$\vdash \exists z\,P(f(a),z,f(f(a)))$

* (c) $\forall y\,Q(b,y),\ \forall x\,\forall y\,(Q(x,y) \to Q(s(x),s(y)))$
$\vdash \exists z\,(Q(b,z) \wedge Q(z,s(s(b))))$

(d) $\forall x\,\forall y\,\forall z\,(S(x,y) \wedge S(y,z) \to S(x,z)),\ \forall x\,\neg S(x,x)$
$\vdash \forall x\,\forall y\,(S(x,y) \to \neg S(y,x))$

(e) $\forall x\,(P(x) \vee Q(x)),\ \exists x\,\neg Q(x),\ \forall x\,(R(x) \to \neg P(x)) \vdash \exists x\,\neg R(x)$

(f) $\forall x\,(P(x) \to (Q(x) \vee R(x))),\ \neg\exists x\,(P(x) \wedge R(x)) \vdash \forall x\,(P(x) \to Q(x))$

(g) $\exists x\,\exists y\,(S(x,y) \vee S(y,x)) \vdash \exists x\,\exists y\,S(x,y)$

(h) $\exists x\,(P(x) \wedge Q(x)),\ \forall y\,(P(x) \to R(x)) \vdash \exists x\,(R(x) \wedge Q(x))$.

14. Translate the following argument into a sequent in predicate logic using a suitable set of predicate symbols:

> If there are any tax payers, then all politicians are tax payers. If there are any philanthropists, then all tax payers are philanthropists. So, if there are any tax-paying philanthropists, then all politicians are philanthropists.

Now come up with a proof of that sequent in predicate logic.

15. Discuss in what sense the equivalences of Theorem 2.13 (page 117) form the basis of an algorithm which, given ϕ, pushes quantifiers to the top of the formula's parse tree. If the result is ψ, what can you say about commonalities and differences between ϕ and ψ?

Exercises 2.4

* 1. Consider the formula $\phi \stackrel{\text{def}}{=} \forall x\, \forall y\, Q(g(x, y), g(y, y), z)$, where Q and g have arity 3 and 2, respectively. Find two models $\mathcal{M}$ and $\mathcal{M}'$ with respective environments l and l' such that $\mathcal{M} \vDash_l \phi$ but $\mathcal{M}' \nvDash_{l'} \phi$.

2. Consider the sentence $\phi \stackrel{\text{def}}{=} \forall x\, \exists y\, \exists z\, (P(x, y) \land P(z, y) \land (P(x, z) \to P(z, x)))$. Which of the following models satisfies ϕ?
   (a) The model $\mathcal{M}$ consists of the set of natural numbers with $P^{\mathcal{M}} \stackrel{\text{def}}{=} \{(m, n) \mid m < n\}$.
   (b) The model $\mathcal{M}'$ consists of the set of natural numbers with $P^{\mathcal{M}'} \stackrel{\text{def}}{=} \{(m, 2 * m) \mid m \text{ natural number}\}$.
   (c) The model $\mathcal{M}''$ consists of the set of natural numbers with $P^{\mathcal{M}''} \stackrel{\text{def}}{=} \{(m, n) \mid m < n + 1\}$.
3. Let P be a predicate with two arguments. Find a model which satisfies the sentence $\forall x\, \neg P(x, x)$; also find one which doesn't.
4. Consider the sentence $\forall x(\exists y P(x, y) \land (\exists z P(z, x) \to \forall y P(x, y)))$. Please simulate the evaluation of this sentence in a model and look-up table of your choice, focusing on how the initial look-up table l grows and shrinks like a stack when you evaluate its subformulas according to the definition of the satisfaction relation.
5. Let ϕ be the sentence $\forall x\, \forall y\, \exists z\, (R(x, y) \to R(y, z))$, where R is a predicate symbol of two arguments.
   * (a) Let $A \stackrel{\text{def}}{=} \{a, b, c, d\}$ and $R^{\mathcal{M}} \stackrel{\text{def}}{=} \{(b, c), (b, b), (b, a)\}$. Do we have $\mathcal{M} \vDash \phi$? Justify your answer, whatever it is.
   * (b) Let $A' \stackrel{\text{def}}{=} \{a, b, c\}$ and $R^{\mathcal{M}'} \stackrel{\text{def}}{=} \{(b, c), (a, b), (c, b)\}$. Do we have $\mathcal{M}' \vDash \phi$? Justify your answer, whatever it is.

* 6. Consider the three sentences

$$\begin{aligned}
\phi_1 &\stackrel{\text{def}}{=} \forall x\, P(x, x) \\
\phi_2 &\stackrel{\text{def}}{=} \forall x\, \forall y\, (P(x, y) \to P(y, x)) \\
\phi_3 &\stackrel{\text{def}}{=} \forall x\, \forall y\, \forall z\, ((P(x, y) \land P(y, z) \to P(x, z)))
\end{aligned}$$

which express that the binary predicate P is reflexive, symmetric and transitive, respectively. Show that none of these sentences is semantically entailed by the other ones by choosing for each pair of sentences above a model which satisfies these two, but not the third sentence – essentially, you are asked to find three binary relations, each satisfying just two of these properties.

7. Show the semantic entailment $\forall x \neg\phi \vDash \neg\exists x\, \phi$; for that you have to take any model which satisfies $\forall x \neg\phi$ and you have to reason why this model must also satisfy $\neg\exists x\, \phi$. You should do this in a similar way to the examples in Section 2.4.2.

* 8. Show the semantic entailment $\forall x\, P(x) \lor \forall x\, Q(x) \vDash \forall x\, (P(x) \lor Q(x))$.

9. Let ϕ and ψ and η be sentences of predicate logic.
   (a) If ψ is semantically entailed by ϕ, is it necessarily the case that ψ is not semantically entailed by $\neg\phi$?
   * (b) If ψ is semantically entailed by $\phi \land \eta$, is it necessarily the case that ψ is semantically entailed by ϕ and semantically entailed by η?
   (c) If ψ is semantically entailed by ϕ or by η, is it necessarily the case that ψ is semantically entailed by $\phi \lor \eta$?
   (d) Explain why ψ is semantically entailed by ϕ iff $\phi \to \psi$ is valid.

10. Is $\forall x\, (P(x) \lor Q(x)) \vDash \forall x\, P(x) \lor \forall x\, Q(x)$ a semantic entailment? Justify your answer.

11. For each set of formulas below show that they are consistent:
   (a) $\forall x \neg S(x, x),\ \exists x\, P(x),\ \forall x\, \exists y\, S(x, y),\ \forall x\, (P(x) \to \exists y\, S(y, x))$
   * (b) $\forall x \neg S(x, x),\ \forall x\, \exists y\, S(x, y),$
   $\forall x\, \forall y\, \forall z\, ((S(x, y) \land S(y, z)) \to S(x, z))$
   (c) $(\forall x\, (P(x) \lor Q(x))) \to \exists y\, R(y),\ \forall x\, (R(x) \to Q(x)),\ \exists y\, (\neg Q(y) \land P(y))$
   * (d) $\exists x\, S(x, x),\ \forall x\, \forall y\, (S(x, y) \to (x = y))$.

12. For each of the formulas of predicate logic below, either find a model which does not satisfy it, or prove it is valid:
   (a) $(\forall x\, \forall y\, (S(x, y) \to S(y, x))) \to (\forall x \neg S(x, x))$
   * (b) $\exists y\, ((\forall x\, P(x)) \to P(y))$
   (c) $(\forall x\, (P(x) \to \exists y\, Q(y))) \to (\forall x\, \exists y\, (P(x) \to Q(y)))$
   (d) $(\forall x\, \exists y\, (P(x) \to Q(y))) \to (\forall x\, (P(x) \to \exists y\, Q(y)))$
   (e) $\forall x\, \forall y\, (S(x, y) \to (\exists z\, (S(x, z) \land S(z, y))))$
   (f) $(\forall x\, \forall y\, (S(x, y) \to (x = y))) \to (\forall z \neg S(z, z))$
   * (g) $(\forall x\, \exists y\, (S(x, y) \land ((S(x, y) \land S(y, x)) \to (x = y)))) \to$
   $(\neg\exists z\, \forall w\, (S(z, w)))$.
   (h) $\forall x\, \forall y\, ((P(x) \to P(y)) \land (P(y) \to P(x)))$
   (i) $(\forall x\, ((P(x) \to Q(x)) \land (Q(x) \to P(x)))) \to ((\forall x\, P(x)) \to (\forall x\, Q(x)))$
   (j) $((\forall x\, P(x)) \to (\forall x\, Q(x))) \to (\forall x\, ((P(x) \to Q(x)) \land (Q(x) \to P(x))))$
   (k) Difficult: $(\forall x\, \exists y\, (P(x) \to Q(y))) \to (\exists y\, \forall x\, (P(x) \to Q(y)))$.

Exercises 2.5

1. Assuming that our proof calculus for predicate logic is sound (see exercise 3 below), show that the validity of the following sequents cannot be proved by finding for each sequent a model such that all formulas to the left of $\vdash$ evaluate to T and the sole formula to the right of $\vdash$ evaluates to F (explain why this guarantees the non-existence of a proof):

(a) $\forall x\,(P(x) \lor Q(x)) \vdash \forall x\,P(x) \lor \forall x\,Q(x)$

* (b) $\forall x\,(P(x) \to R(x)),\ \forall x\,(Q(x) \to R(x)) \vdash \exists x\,(P(x) \land Q(x))$

(c) $(\forall x\,P(x)) \to L \vdash \forall x\,(P(x) \to L)$, where L has arity 0

* (d) $\forall x\,\exists y\,S(x,y) \vdash \exists y\,\forall x\,S(x,y)$

(e) $\exists x\,P(x),\ \exists y\,Q(y) \vdash \exists z\,(P(z) \land Q(z))$.

* (f) $\exists x\,(\neg P(x) \land Q(x)) \vdash \forall x\,(P(x) \to Q(x))$

* (g) $\exists x\,(\neg P(x) \lor \neg Q(x)) \vdash \forall x\,(P(x) \lor Q(x))$.

2. Assuming that $\vdash$ is sound and complete for $\vDash$ in first-order logic, explain in detail why the undecidability of $\vDash$ implies that satisfiability, validity, and provability are all undecidable for that logic.
3. To show the soundness of our natural deduction rules for predicate logic, it intuitively suffices to show that the conclusion of a proof rule is true provided that all its premises are true. What additional complication arises due to the presence of variables and quantifiers? Can you precisely formalise the necessary induction hypothesis for proving soundness?

Exercises 2.6

1. In Example 2.23, page 136, does $\mathcal{M} \vDash_l \exists P\,\phi$ hold if l satisfies

   * (a) $l(u) = s_3$ and $l(v) = s_1$;

   (b) $l(u) = s_1$ and $l(v) = s_3$?

   Justify your answers.
2. Prove that $\mathcal{M} \vDash_l \exists P\,\forall x \forall y \forall z\,(C_1 \land C_2 \land C_3 \land C_4)$ holds iff state $l(v)$ is not reachable from state $l(u)$ in the model $\mathcal{M}$, where the C_i are the ones of (2.12) on page 139.
3. Does Theorem 2.26 from page 138 apply or remain valid if we allow ϕ to contain function symbols of any finite arity?

* 4. In the directed graph of Figure 2.5 from page 137, how many paths are there that witness the reachability of node s_3 from s_2?

5. Let P and R be predicate symbols of arity 2. Write formulas of existential second-order logic of the form $\exists P\,\psi$ that hold in all models of the form $\mathcal{M} = (A, R^{\mathcal{M}})$ iff

   * (a) R contains a reflexive and symmetric relation;

   (b) R contains an equivalence relation

   (c) there is an R-path that visits each node of the graph exactly once – such a path is called Hamiltonian

   (d) R can be extended to an equivalence relation: there is some equivalence relation T with $R^{\mathcal{M}} \subseteq T$

   * (e) the relation 'there is an R-path of length 2' is transitive.

* 6. Show informally that (2.16) on page 141 gives rise to Russell's paradox: A has to be, and cannot be, an element of A.

7. The second item in the proof of Theorem 2.28 (page 140) relies on the fact that if a binary relation R is contained in a reflexive, transitive relation T of

the same type, then T also contains the reflexive, transitive closure of R. Prove this.

8. For the model of Example 2.23 and Figure 2.5 (page 137), determine which model checks hold and justify your answer:
* (a) $\exists P\,(\forall x \forall y\, P(x, y) \to \neg P(y, x)) \land (\forall u \forall v\, R(u, v) \to P(v, u))$;
(b) $\forall P\,(\exists x \exists y \exists z\, P(x, y) \land P(y, z) \land \neg P(x, z)) \to (\forall u \forall v\, R(u, v) \to P(u, v))$; and
(c) $\forall P\,(\forall x\, \neg P(x, x)) \lor (\forall u \forall v\, R(u, v) \to P(u, v))$.

9. Express the following statements about a binary relation R in predicate logic, universal second-order logic, or existential second-order logic – if at all possible:
(a) All symmetric, transitive relations either don't contain R or are equivalence relations.
* (b) All nodes are on at least one R-cycle.
(c) There is a smallest relation containing R which is symmetric.
(d) There is a smallest relation containing R which is reflexive.
* (e) The relation R is a maximal equivalence relation: R is an equivalence relation; and there is no relation contained in R that is an equivalence relation.

Exercises 2.7

1\.* (a) Explain why the model of Figure 2.11 (page 148) is a counterexample to `OfLovers` in the presence of the fact `NoSelfLove`.
(b) Can you identify the set $\{a, b, c\}$ from Example 2.19 (page 128) with the model of Figure 2.11 such that these two models are structurally the same? Justify your answer.
* (c) Explain informally why no model with less than three elements can satisfy (2.8) from page 128 and the fact `NoSelfLove`.

2. Use the following fragment of an Alloy module

```
module AboutGraphs

sig Element {}

sig Graph {
  nodes : set Element,
  edges : nodes -> nodes
}
```

for these modelling tasks:
(a) Recall Exercise 6 from page 163 and its three sentences, where $P(x, y)$ specifies that there is an edge from x to y. For each sentence, write a consistency check that attempts to generate a model of a graph in which that sentence is false, but the other two are true. Analyze it within Alloy. What it the smallest scope, if any, in which the analyzer finds a model for this?

* (b) (Recall that the expression `# S = n` specifies that set `S` has n elements.) Use Alloy to generate a graph with seven nodes such that each node can reach exactly five nodes on finite paths (not necessarily the same five nodes).

(c) A cycle of length n is a set of n nodes and a path through each of them, beginning and ending with the same node. Generate a cycle of length 4.

3. An undirected graph has a set of nodes and a set of edges, except that every edge connects two nodes without any sense of direction.

   (a) Adjust the Alloy module from the previous item – e.g. by adding an appropriate `fact` – to 'simulate' undirected graphs.

   (b) Write some consistency and assertion checks and analyze them to boost the confidence you may have in your Alloy module of undirected graphs.

* 4. A colorable graph consists of a set of nodes, a binary symmetric relation (the edges) between nodes and a function that assigns to each node a color. This function is subject to the constraint that no nodes have the same color if they are related by an edge.

   (a) Write a signature `AboutColoredGraphs` for this structure and these constraints.

   (b) Write a `fun`-statement that generates a graph whose nodes are colored by two colors only. Such a graph is 2-colorable.

   (c) For eack $k = 3, 4$ write a `fun`-statement that generates a graph whose nodes are colored by k colors such that all k colors are being used. Such a graph is k-colorable.

   (d) Test these three functions in a module.

   (e) Try to write a `fun`-statement that generates a graph that is 3-colorable but definitely not 2-colorable. What does Alloy's model builder report? Consider the formula obtained from that `fun`-statement's body by existentially quantifying that body with all its parameters. Determine whether is belongs to predicate logic, existential or universal second-order logic.

* 5. A Kripke model is a state machine with a non-empty set of initial states `init`, a mapping `prop` from states to atomic properties (specifying which properties are true at which states), a state transition relation `next`, and a set of final states `final` (states that don't have a next state). With a module `KripkeModel`:

   (a) Write a signature `StateMachine` and some basic facts that reflect this structure and these constraints.

   (b) Write a `fun`-statement `Reaches` which takes a state machine as first parameter and a set of states as a second parameter such that the second parameter denotes the first parameter's set of states reachable from any initial state. Note: Given the type declaration `r : T -> T`, the expression `*r` has type `T -> T` as well and denotes the reflexive, transitive closure of `r`.

   (c) Write these `fun`-statements and check their consistency:

      i. `DeadlockFree(m: StateMachine)`, among the reachable states of `m` only the `final` ones can deadlock;

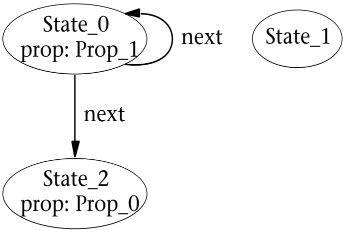


Figure 2.15. A snapshot of a non-deterministic state machine in which no non-final state deadlocks and where states that satisfy the same properties are identical.

ii. `Deterministic(m: StateMachine)`, at all reachable states of `m` the state transition relation is deterministic: each state has at most one outgoing transition;

iii. `Reachability(m: StateMachine, p: Prop)`, some state which has property `p` can be reached in `m`; and

iv. `Liveness(m: StateMachine, p: Prop)`, no matter which state `m` reaches, it can – from that state – reach a state in which `p` holds.

(d) i. Write an assertion `Implies` which says that whenever a state machine satisfies `Liveness` for a property then it also satisfies `Reachability` for that property.

ii. Analyze that assertion in a scope of your choice. What conclusions can you draw from the analysis' findings?

(e) Write an assertion `Converse` which states that `Reachability` of a property implies its `Liveness`. Analyze it in a scope of 3. What do you conclude, based on the analysis' result?

(f) Write a `fun`-statement that, when analyzed, generates a statemachine with two propositions and three states such that it satisfies the statement of the sentence in the caption of Figure 2.15.

* 6. Groups are the bread and butter of cryptography and group operations are applied in the silent background when you use PUTTY, Secure Socket Layers etc. A group is a tuple $(G, \star, 1)$, where $\star\colon G \times G \to G$ is a function and $1 \in G$ such that

G1 for every $x \in G$ there is some $y \in G$ such that $x \star y = y \star x = 1$ (any such y is called an inverse of x);

G2 for all $x, y, z \in G$, we have $x \star (y \star z) = (x \star y) \star z$; and

G3 for all $x \in G$, we have $x \star 1 = 1 \star x = x$.

(a) Specify a signature for groups that realizes this functionality and its constraints.

(b) Write a `fun`-statement `AGroup` that generates a group with three elements.

(c) Write an assertion `Inverse` saying that inverse elements are unique. Check it in the scope of 5. Report your findings. What would the small scope hypothesis suggest?

(d) i. Write an assertion Commutative saying that all groups are commutative. A group is commutative iff $x \star y = y \star x$ for all its elements x and y.

ii. Check the assertion Commutative in scope 5 and report your findings. What would the small scope hypothesis suggest?

iii. Re-check assertion Commutative in scope 6 and record how long the tool takes to find a solution. What lesson(s) do you learn from this?

(e) For the functions and assertions above, is it safe to restrict the scope for groups to 1? And how does one do this in Alloy?

7. In Alloy, one can extend a signature. For example, we may declare

```
sig Program extends PDS {
  m : components   -- initial main of PDS
}
```

This declares instances of Program to be of type PDS, but to also possess a designated component named m. Observe how the occurrence of components in m : components refers to the set of components of a program, viewed as a PDS[5]. In this exercise, you are asked to modify the Alloy module of Figure 2.13 on page 154.

(a) Include a signature Program as above. Add a fact stating that all programs' designated component has a main method; and for all programs, their set of components is the reflexive, transitive closure of their relation requires applied to the designated component m. Alloy uses *r to denote the reflexive, transitive closure of relation r.

(b) Write a guided simulation that, if consistent, produces a model with three PDSs, exactly one of them being a program. The program has four components – including the designated m – all of which schedule services from the remaining three components. Use Alloy's analyzer to detemine whether your simulation is consistent and compliant with the specification given in this item.

(c) Let's say that a component of a program is garbage for that program if no service reachable from the main service of m via requires schedules that component. Explain whether, and if so how, the constraints of AddComponent and RemoveComponent already enforce the presence of 'garbage collection' if the instances of P and P' are constrained to be programs.

8. Recall our discussion of existential and universal second-order logic from Section 2.6. Then study the structure of the fun-statements and assertions in Figure 2.13 on page 154. As you may know, Alloy analyzes such statements by deriving from them a formula for which it tries to find a model within the specified scope: the negation of the body of an assertion; or the body of a fun-statement, existentially quantified with all its parameters. For each of these derived formulas,

[5] In most object-oriented languages, e.g. Java, extends creates a new type. In Alloy 2.0 and 2.1, it creates a subset of a type and not a new type as such, where the subset has additional structure and may need to satisfy additional constraints.

determine whether they can be expressed in first-order logic, existential second-order logic or universal second-order logic.

9. Recalling the comment on page 142 that Alloy combines model checking $\mathcal{M} \vDash \phi$ and validity checking $\Gamma \vDash \phi$, can you discuss to what extent this is so?

2.9 Bibliographic notes

Many design decisions have been taken in the development of predicate logic in the form known today. The Greeks and the medievals had systems in which many of the examples and exercises in this book could be represented, but nothing that we would recognise as predicate logic emerged until the work of Gottlob Frege in 1879, printed in [Fre03]. An account of the contributions of the many other people involved in the development of logic can be found in the first few pages of W. Hodges' chapter in [Hod83].

There are many books covering classical logic and its use in computer science; we give a few incomplete pointers to the literature. The books [SA91], [vD89] and [Gal87] cover more theoretical applications than those in this book, including type theory, logic programming, algebraic specification and term-rewriting systems. An approach focusing on automatic theorem proving is taken by [Fit96]. Books which study the mathematical aspects of predicate logic in greater detail, such as completeness of the proof systems and incompleteness of first-order arithmetic, include [Ham78] and [Hod83].

Most of these books present other proof systems besides natural deduction such as axiomatic systems and tableau systems. Although natural deduction has the advantages of elegance and simplicity over axiomatic methods, there are few expositions of it in logic books aimed at a computer science audience. One exception to this is the book [BEKV94], which is the first one to present the rules for quantifiers in the form we used here. A natural deduction theorem prover called Jape has been developed, in which one can vary the set of available rules and specify new ones[6].

A standard reference for computability theory is [BJ80]. A proof for the undecidability of the Post correspondence problem can be found in the text book [Tay98]. The second instance of a Post correspondence problem is taken from [Sch92]. A text on the fundamentals of databases systems is [EN94]. The discussion of Section 2.6 is largely based on the text [Pap94] which we highly recommend if you mean to find out more about the intimate connections between logic and computational complexity.

[6] www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/jape.html

The source code of all complete Alloy modules from this chapter (working under Alloy 2.0 and 2.1) as well as source code compliant with Alloy 3.0 are available under 'ancillary material' at the book's website. The PDS model grew out of a coursework set in the Fall 2002 for *C475 Software Engineering Environments*, co-taught by Susan Eisenbach and the first author; a published model customized for the .NET global assembly cache will appeared in [EJC03]. The modelling language Alloy and its constraint analyzer [JSS01] have been developed by D. Jackson and his Software Design Group at the Laboratory for Computer Science at the Massachusetts Institute of Technology. The tool has a dedicated repository website at `alloy.mit.edu`.

More information on typed higher-order logics and their use in the modelling and verifying of programming frameworks can be found on F. Pfenning's course homepage[7] on Computation and Deduction.

[7] `www-2.cs.cmu.edu/~fp/courses/comp-ded/`

3

Verification by model checking

3.1 Motivation for verification

There is a great advantage in being able to verify the correctness of computer systems, whether they are hardware, software, or a combination. This is most obvious in the case of *safety-critical systems*, but also applies to those that are *commercially critical*, such as mass-produced chips, *mission critical*, etc. Formal verification methods have quite recently become usable by industry and there is a growing demand for professionals able to apply them. In this chapter, and the next one, we examine two applications of logics to the question of verifying the correctness of computer systems, or programs.

Formal verification techniques can be thought of as comprising three parts:

- a *framework for modelling systems*, typically a description language of some sort;
- a *specification language* for describing the properties to be verified;
- a *verification method* to establish whether the description of a system satisfies the specification.

Approaches to verification can be classified according to the following criteria:

Proof-based vs. model-based. In a proof-based approach, the system description is a set of formulas Γ (in a suitable logic) and the specification is another formula ϕ. The verification method consists of trying to find a proof that $\Gamma \vdash \phi$. This typically requires guidance and expertise from the user.

In a model-based approach, the system is represented by a model $\mathcal{M}$ for an appropriate logic. The specification is again represented by a formula ϕ and the verification method consists of computing whether a model $\mathcal{M}$ satisfies ϕ (written $\mathcal{M} \vDash \phi$). This computation is usually automatic for finite models.

In Chapters 1 and 2, we could see that logical proof systems are often sound and complete, meaning that $\Gamma \vdash \phi$ (provability) holds if, and only if, $\Gamma \vDash \phi$ (semantic entailment) holds, where the latter is defined as follows: for all models $\mathcal{M}$, if for all $\psi \in \Gamma$ we have $\mathcal{M} \vDash \psi$, then $\mathcal{M} \vDash \phi$. Thus, we see that the model-based approach is potentially simpler than the proof-based approach, for it is based on a single model $\mathcal{M}$ rather than a possibly infinite class of them.

Degree of automation. Approaches differ on how automatic the method is; the extremes are fully automatic and fully manual. Many of the computer-assisted techniques are somewhere in the middle.

Full- vs. property-verification. The specification may describe a single property of the system, or it may describe its full behaviour. The latter is typically expensive to verify.

Intended domain of application, which may be hardware or software; sequential or concurrent; reactive or terminating; etc. A reactive system is one which reacts to its environment and is not meant to terminate (e.g., operating systems, embedded systems and computer hardware).

Pre- vs. post-development. Verification is of greater advantage if introduced early in the course of system development, because errors caught earlier in the production cycle are less costly to rectify. (It is alleged that Intel lost millions of dollars by releasing their Pentium chip with the FDIV error.)

This chapter concerns a verification method called *model checking.* In terms of the above classification, model checking is an automatic, model-based, property-verification approach. It is intended to be used for *concurrent, reactive* systems and originated as a post-development methodology. Concurrency bugs are among the most difficult to find by *testing* (the activity of running several simulations of important scenarios), since they tend to be non-reproducible or not covered by test cases, so it is well worth having a verification technique that can help one to find them.

The Alloy system described in Chapter 2 is also an automatic, model-based, property-verification approach. The way models are used is slightly different, however. Alloy finds models which form counterexamples to assertions made by the user. Model checking starts with a model described by the user, and discovers whether hypotheses asserted by the user are valid on the model. If they are not, it can produce counterexamples, consisting of execution traces. Another difference between Alloy and model checking is that model checking (unlike Alloy) focuses explicitly on temporal properties and the temporal evolution of systems.

By contrast, Chapter 4 describes a very different verification technique which in terms of the above classification is a proof-based, computer-assisted, property-verification approach. It is intended to be used for programs which we expect to terminate and produce a result.

Model checking is based on *temporal logic*. The idea of temporal logic is that a formula is not *statically* true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others. Thus, the static notion of truth is replaced by a *dynamic* one, in which the formulas may change their truth values as the system evolves from state to state. In model checking, the models $\mathcal{M}$ are *transition systems* and the properties ϕ are formulas in temporal logic. To verify that a system satisfies a property, we must do three things:

- model the system using the description language of a model checker, arriving at a model $\mathcal{M}$;
- code the property using the specification language of the model checker, resulting in a temporal logic formula ϕ;
- Run the model checker with inputs $\mathcal{M}$ and ϕ.

The model checker outputs the answer 'yes' if $\mathcal{M} \vDash \phi$ and 'no' otherwise; in the latter case, most model checkers also produce a trace of system behaviour which causes this failure. This automatic generation of such 'counter traces' is an important tool in the design and debugging of systems.

Since model checking is a *model-based* approach, in terms of the classification given earlier, it follows that in this chapter, unlike in the previous two, we will not be concerned with semantic entailment ($\Gamma \vDash \phi$), or with proof theory ($\Gamma \vdash \phi$), such as the development of a natural deduction calculus for temporal logic. We will work solely with the notion of satisfaction, i.e. the satisfaction relation between a model and a formula ($\mathcal{M} \vDash \phi$).

There is a whole zoo of temporal logics that people have proposed and used for various things. The abundance of such formalisms may be organised by classifying them according to their particular view of 'time.' *Linear-time* logics think of time as a set of paths, where a path is a sequence of time instances. *Branching-time* logics represent time as a tree, rooted at the present moment and branching out into the future. Branching time appears to make the non-deterministic nature of the future more explicit. Another quality of time is whether we think of it as being *continuous* or *discrete*. The former would be suggested if we study an analogue computer, the latter might be preferred for a synchronous network.

Temporal logics have a dynamic aspect to them, since the truth of a formula is not fixed in a model, as it is in predicate or propositional logic, but depends on the time-point inside the model. In this chapter, we study a logic where time is linear, called *Linear-time Temporal Logic* (LTL), and another where time is branching, namely *Computation Tree Logic* (CTL). These logics have proven to be extremely fruitful in verifying hardware and communication protocols; and people are beginning to apply them to the verification of software. *Model checking* is the process of computing an answer to the question of whether $\mathcal{M}, s \vDash \phi$ holds, where ϕ is a formula of one of these logics, $\mathcal{M}$ is an appropriate model of the system under consideration, s is a state of that model and $\vDash$ is the underlying satisfaction relation.

Models like $\mathcal{M}$ should not be confused with an actual physical system. Models are abstractions that omit lots of real features of a physical system, which are irrelevant to the checking of ϕ. This is similar to the abstractions that one does in calculus or mechanics. There we talk about *straight* lines, *perfect* circles, or an experiment *without friction*. These abstractions are very powerful, for they allow us to focus on the essentials of our particular concern.

3.2 Linear-time temporal logic

Linear-time temporal logic, or LTL for short, is a temporal logic, with connectives that allow us to refer to the future. It models time as a sequence of states, extending infinitely into the future. This sequence of states is sometimes called a computation path, or simply a path. In general, the future is not determined, so we consider several paths, representing different possible futures, any one of which might be the 'actual' path that is realised.

We work with a fixed set `Atoms` of atomic formulas (such as $p, q, r, \dots$, or $p_1, p_2, \dots$). These atoms stand for atomic facts which may hold of a system, like *'Printer Q5 is busy,'* or *'Process 3259 is suspended,'* or *'The content of register* `R1` *is the integer value* 6.*'* The choice of atomic descriptions obviously depends on our particular interest in a system at hand.

3.2.1 Syntax of LTL

Definition 3.1 Linear-time temporal logic (LTL) has the following syntax given in Backus Naur form:

$$\begin{aligned} \phi ::= {} & \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \\ & \mid (\mathrm{X}\,\phi) \mid (\mathrm{F}\,\phi) \mid (\mathrm{G}\,\phi) \mid (\phi\ \mathrm{U}\ \phi) \mid (\phi\ \mathrm{W}\ \phi) \mid (\phi\ \mathrm{R}\ \phi) \end{aligned} \tag{3.1}$$

where p is any propositional atom from some set `Atoms`.

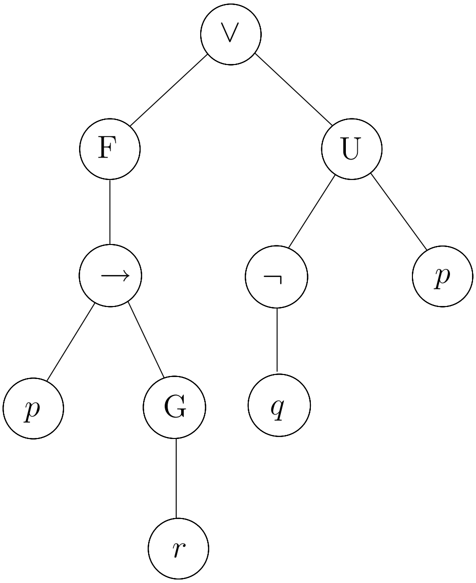


Figure 3.1. The parse tree of $(\mathrm{F}\,(p \rightarrow \mathrm{G}\,r) \vee ((\neg q)\ \mathrm{U}\ p))$.

Thus, the symbols $\top$ and $\bot$ are LTL formulas, as are all atoms from `Atoms`; and $\neg\phi$ is an LTL formula if ϕ is one, etc. The connectives X, F, G, U, R, and W are called *temporal connectives.* X means 'neXt state,' F means 'some Future state,' and G means 'all future states (Globally).' The next three, U, R and W are called 'Until,' 'Release' and 'Weak-until' respectively. We will look at the precise meaning of all these connectives in the next section; for now, we concentrate on their syntax.

Here are some examples of LTL formulas:

- $(((\mathrm{F}\,p) \wedge (\mathrm{G}\,q)) \rightarrow (p\ \mathrm{W}\ r))$
- $(\mathrm{F}\,(p \rightarrow (\mathrm{G}\,r)) \vee ((\neg q)\ \mathrm{U}\ p))$, the parse tree of this formula is illustrated in Figure 3.1.
- $(p\ \mathrm{W}\ (q\ \mathrm{W}\ r))$
- $((\mathrm{G}\,(\mathrm{F}\,p)) \rightarrow (\mathrm{F}\,(q \vee s)))$.

It's boring to write all those brackets, and makes the formulas hard to read. Many of them can be omitted without introducing ambiguities; for example, $(p \rightarrow (\mathrm{F}\,q))$ could be written $p \rightarrow \mathrm{F}\,q$ without ambiguity. Others, however, are required to resolve ambiguities. In order to omit some of those, we assume similar binding priorities for the LTL connectives to those we assumed for propositional and predicate logic.

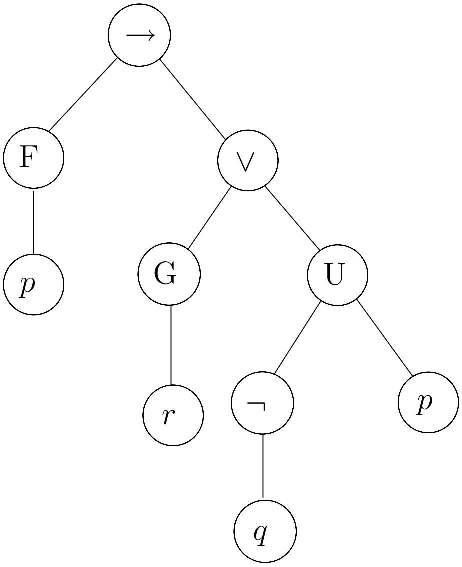


Figure 3.2. The parse tree of $\mathrm{F}\,p \to \mathrm{G}\,r \vee \neg q\ \mathrm{U}\ p$, assuming binding priorities of Convention 3.2.

Convention 3.2 The unary connectives (consisting of $\neg$ and the temporal connectives X, F and G) bind most tightly. Next in the order come U, R and W; then come $\wedge$ and $\vee$; and after that comes $\to$.

These binding priorities allow us to drop some brackets without introducing ambiguity. The examples above can be written:

- $\mathrm{F}\,p \wedge \mathrm{G}\,q \to p\ \mathrm{W}\ r$
- $\mathrm{F}\,(p \to \mathrm{G}\,r) \vee \neg q\ \mathrm{U}\ p$
- $p\ \mathrm{W}\ (q\ \mathrm{W}\ r)$
- $\mathrm{G}\,\mathrm{F}\,p \to \mathrm{F}\,(q \vee s)$.

The brackets we retained were in order to override the priorities of Convention 3.2, or to disambiguate cases which the convention does not resolve. For example, with no brackets at all, the second formula would become $\mathrm{F}\,p \to \mathrm{G}\,r \vee \neg q\ \mathrm{U}\ p$, corresponding to the parse tree of Figure 3.2, which is quite different.

The following are *not* well-formed formulas:

- $\mathrm{U}\,r$ – since U is binary, not unary
- $p\ \mathrm{G}\ q$ – since G is unary, not binary.

Definition 3.3 A subformula of an LTL formula ϕ is any formula ψ whose parse tree is a subtree of ϕ's parse tree.

The subformulas of $p \; \mathrm{W} \; (q \; \mathrm{U} \; r)$, e.g., are p, q, r, $q \; \mathrm{U} \; r$ and $p \; \mathrm{W} \; (q \; \mathrm{U} \; r)$.

3.2.2 Semantics of LTL

The kinds of systems we are interested in verifying using LTL may be modelled as transition systems. A transition system models a system by means of *states* (static structure) and *transitions* (dynamic structure). More formally:

Definition 3.4 A transition system $\mathcal{M} = (S, \rightarrow, L)$ is a set of states S endowed with a transition relation $\rightarrow$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$, and a labelling function $L \colon S \rightarrow \mathcal{P}(\mathtt{Atoms})$.

Transition systems are also simply called *models* in this chapter. So a model has a collection of states S, a relation $\rightarrow$, saying how the system can move from state to state, and, associated with each state s, one has the set of atomic propositions $L(s)$ which are true at that particular state. We write $\mathcal{P}(\mathtt{Atoms})$ for the power set of $\mathtt{Atoms}$, a collection of atomic descriptions. For example, the power set of $\{p, q\}$ is $\{\emptyset, \{p\}, \{q\}, \{p, q\}\}$. A good way of thinking about L is that it is just an assignment of truth values to all the propositional atoms, as it was the case for propositional logic (we called that a *valuation*). The difference now is that we have *more than one state*, so this assignment depends on which state s the system is in: $L(s)$ contains all atoms which are true in state s.

We may conveniently express all the information about a (finite) transition system $\mathcal{M}$ using directed graphs whose nodes (which we call states) contain all propositional atoms that are true in that state. For example, if our system has only three states s_0, s_1 and s_2; if the only possible transitions between states are $s_0 \rightarrow s_1$, $s_0 \rightarrow s_2$, $s_1 \rightarrow s_0$, $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_2$; and if $L(s_0) = \{p, q\}$, $L(s_1) = \{q, r\}$ and $L(s_2) = \{r\}$, then we can condense all this information into Figure 3.3. We prefer to present models by means of such pictures whenever that is feasible.

The requirement in Definition 3.4 that for every $s \in S$ there is at least one $s' \in S$ such that $s \rightarrow s'$ means that no state of the system can 'deadlock.' This is a technical convenience, and in fact it does not represent any real restriction on the systems we can model. If a system did deadlock, we could always add an extra state s_d representing deadlock, together with new

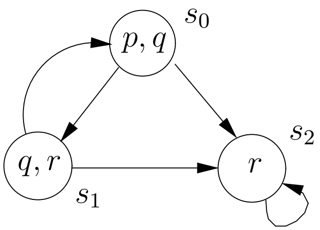


Figure 3.3. A concise representation of a transition system $\mathcal{M} = (S, \to, L)$ as a directed graph. We label state s with l iff $l \in L(s)$.

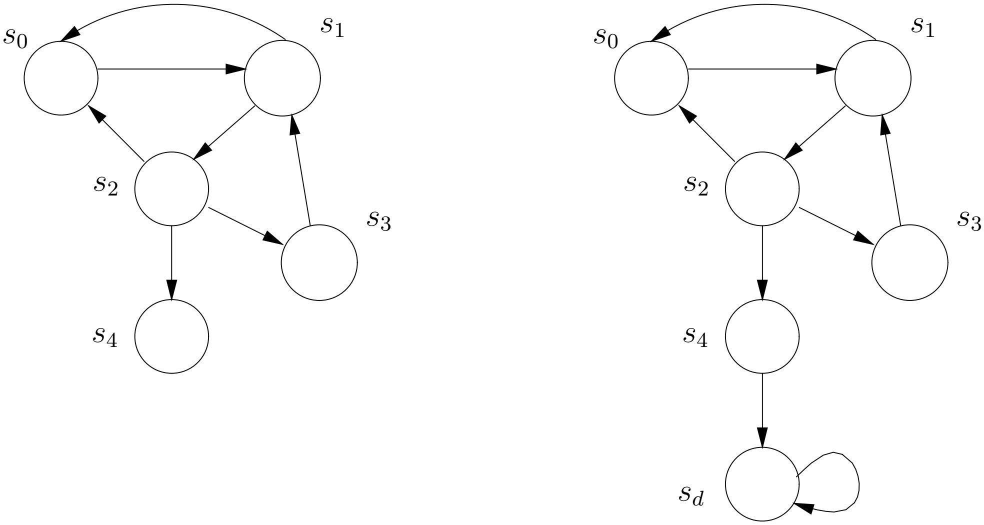


Figure 3.4. On the left, we have a system with a state s_4 that does not have any further transitions. On the right, we expand that system with a 'deadlock' state s_d such that no state can deadlock; of course, it is then our understanding that reaching the 'deadlock' state s_d corresponds to deadlock in the original system.

transitions $s \to s_d$ for each s which was a deadlock in the old system, as well as $s_d \to s_d$. See Figure 3.4 for such an example.

Definition 3.5 A path in a model $\mathcal{M} = (S, \to, L)$ is an infinite sequence of states $s_1, s_2, s_3, \dots$ in S such that, for each $i \geq 1$, $s_i \to s_{i+1}$. We write the path as $s_1 \to s_2 \to \dots$.

Consider the path $\pi = s_1 \to s_2 \to \dots$. It represents a possible future of our system: first it is in state s_1, then it is in state s_2, and so on. We write π^i for the suffix starting at s_i, e.g., π^3 is $s_3 \to s_4 \to \dots$.

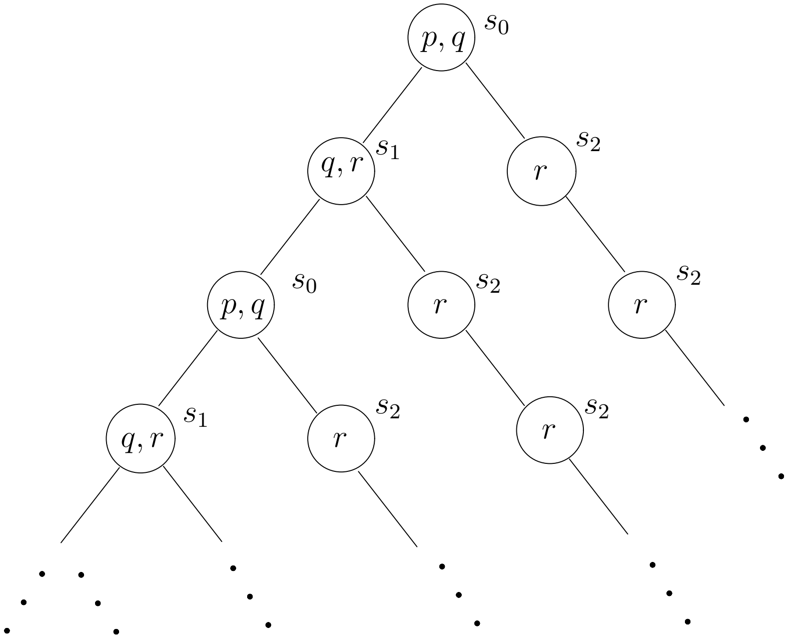


Figure 3.5. Unwinding the system of Figure 3.3 as an infinite tree of all computation paths beginning in a particular state.

It is useful to visualise all possible computation paths from a given state s by unwinding the transition system to obtain an infinite computation tree. For example, if we unwind the state graph of Figure 3.3 for the designated starting state s_0, then we get the infinite tree in Figure 3.5. The execution paths of a model $\mathcal{M}$ are explicitly represented in the tree obtained by unwinding the model.

Definition 3.6 Let $\mathcal{M} = (S, \rightarrow, L)$ be a model and $\pi = s_1 \rightarrow \ldots$ be a path in $\mathcal{M}$. Whether π satisfies an LTL formula is defined by the satisfaction relation $\vDash$ as follows:

1. $\pi \vDash \top$
2. $\pi \nvDash \bot$
3. $\pi \vDash p$ iff $p \in L(s_1)$
4. $\pi \vDash \neg\phi$ iff $\pi \nvDash \phi$
5. $\pi \vDash \phi_1 \wedge \phi_2$ iff $\pi \vDash \phi_1$ and $\pi \vDash \phi_2$
6. $\pi \vDash \phi_1 \vee \phi_2$ iff $\pi \vDash \phi_1$ or $\pi \vDash \phi_2$
7. $\pi \vDash \phi_1 \rightarrow \phi_2$ iff $\pi \vDash \phi_2$ whenever $\pi \vDash \phi_1$
8. $\pi \vDash \mathrm{X}\,\phi$ iff $\pi^2 \vDash \phi$
9. $\pi \vDash \mathrm{G}\,\phi$ iff, for all $i \geq 1$, $\pi^i \vDash \phi$

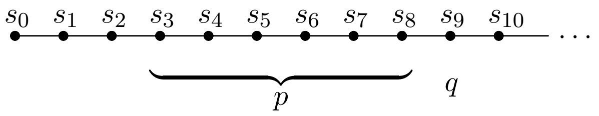

s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10}
...
p
q

Figure 3.6. An illustration of the meaning of Until in the semantics of LTL. Suppose p is satisfied at (and only at) s_3, s_4, s_5, s_6, s_7, s_8 and q is satisfied at (and only at) s_9. Only the states s_3 to s_9 each satisfy $p \mathrm{~U~} q$ along the path shown.

10. $\pi \vDash \mathrm{F}\, \phi$ iff there is some $i \geq 1$ such that $\pi^i \vDash \phi$
11. $\pi \vDash \phi \mathrm{~U~} \psi$ iff there is some $i \geq 1$ such that $\pi^i \vDash \psi$ and for all $j = 1, \dots, i-1$ we have $\pi^j \vDash \phi$
12. $\pi \vDash \phi \mathrm{~W~} \psi$ iff either there is some $i \geq 1$ such that $\pi^i \vDash \psi$ and for all $j = 1, \dots, i-1$ we have $\pi^j \vDash \phi$; or for all $k \geq 1$ we have $\pi^k \vDash \phi$
13. $\pi \vDash \phi \mathrm{~R~} \psi$ iff either there is some $i \geq 1$ such that $\pi^i \vDash \phi$ and for all $j = 1, \dots, i$ we have $\pi^j \vDash \psi$, or for all $k \geq 1$ we have $\pi^k \vDash \psi$.

Clauses 1 and 2 reflect the facts that $\top$ is always true, and $\bot$ is always false. Clauses 3–7 are similar to the corresponding clauses we saw in propositional logic. Clause 8 removes the first state from the path, in order to create a path starting at the 'next' (second) state.

Notice that clause 3 means that atoms are evaluated in the first state along the path in consideration. However, that doesn't mean that all the atoms occuring in an LTL formula refer to the first state of the path; if they are in the scope of a temporal connective, e.g., in $\mathrm{G}\,(p \to \mathrm{X}\, q)$, then the calculation of satisfaction involves taking suffices of the path in consideration, and the atoms refer to the first state of those suffices.

Let's now look at clauses 11–13, which deal with the binary temporal connectives. U, which stands for 'Until,' is the most commonly encountered one of these. The formula $\phi_1 \mathrm{~U~} \phi_2$ holds on a path if it is the case that ϕ_1 holds continuously *until* ϕ_2 holds. Moreover, $\phi_1 \mathrm{~U~} \phi_2$ actually demands that ϕ_2 *does* hold in some future state. See Figure 3.6 for illustration: each of the states s_3 to s_9 satisfies $p \mathrm{~U~} q$ along the path shown, but s_0 to s_2 don't.

The other binary connectives are W, standing for 'Weak-until,' and R, standing for 'Release.' Weak-until is just like U, except that $\phi \mathrm{~W~} \psi$ does not require that ψ is eventually satisfied along the path in question, which is required by $\phi \mathrm{~U~} \psi$. Release R is the dual of U; that is, $\phi \mathrm{~R~} \psi$ is equivalent to $\neg(\neg\phi \mathrm{~U~} \neg\psi)$. It is called 'Release' because clause 11 determines that ψ must remain true up to and including the moment when ϕ becomes true (if there is one); ϕ 'releases' ψ. R and W are actually quite similar; the differences are that they swap the roles of ϕ and ψ, and the clause for W has an $i - 1$

where R has i. Since they are similar, why do we need both? We don't; they are interdefinable, as we will see later. However, it's useful to have both. R is useful because it is the dual of U, while W is useful because it is a weak form of U.

Note that neither the strong version (U) or the weak version (W) of until says anything about what happens after the until has been realised. This is in contrast with some of the readings of 'until' in natural language. For example, in the sentence 'I smoked until I was 22' it is not only expressed that the person referred to continually smoked up until he or she was 22 years old, but we also would interpret such a sentence as saying that this person gave up smoking from that point onwards. This is different from the semantics of until in temporal logic. We could express the sentence about smoking by combining U with other connectives; for example, by asserting that it was once true that $s \, \mathrm{U} \, (t \wedge \mathrm{G} \, \neg s)$, where s represents 'I smoke' and t represents 'I am 22.'

Remark 3.7 Notice that, in clauses 9–13 above, the future includes the present. This means that, when we say 'in all future states,' we are including the present state as a future state. It is a matter of convention whether we do this, or not. As an exercise, you may consider developing a version of LTL in which the future excludes the present. A consequence of adopting the convention that the future shall include the present is that the formulas $\mathrm{G} \, p \to p$, $p \to q \, \mathrm{U} \, p$ and $p \to \mathrm{F} \, p$ are true in every state of every model.

So far we have defined a satisfaction relation between paths and LTL formulas. However, to verify systems, we would like to say that a model as a whole satisfies an LTL formula. This is defined to hold whenever *every* possible execution path of the model satisfies the formula.

Definition 3.8 Suppose $\mathcal{M} = (S, \to, L)$ is a model, $s \in S$, and ϕ an LTL formula. We write $\mathcal{M}, s \vDash \phi$ if, for every execution path π of $\mathcal{M}$ starting at s, we have $\pi \vDash \phi$.

If $\mathcal{M}$ is clear from the context, we may abbreviate $\mathcal{M}, s \vDash \phi$ by $s \vDash \phi$. It should be clear that we have outlined the formal foundations of a procedure that, given ϕ, $\mathcal{M}$ and s, can check whether $\mathcal{M}, s \vDash \phi$ holds. Later in this chapter, we will examine algorithms which implement this calculation. Let us now look at some example checks for the system in Figures 3.3 and 3.5.

1. $\mathcal{M}, s_0 \vDash p \wedge q$ holds since the atomic symbols p and q are contained in the node of s_0: $\pi \vDash p \wedge q$ for *every* path π beginning in s_0.

2. $\mathcal{M}, s_0 \vDash \neg r$ holds since the atomic symbol r is *not* contained in node s_0.
3. $\mathcal{M}, s_0 \vDash \top$ holds by definition.
4. $\mathcal{M}, s_0 \vDash \mathrm{X}\, r$ holds since all paths from s_0 have either s_1 or s_2 as their next state, and each of those states satisfies r.
5. $\mathcal{M}, s_0 \vDash \mathrm{X}\,(q \wedge r)$ does not hold since we have the rightmost computation path $s_0 \to s_2 \to s_2 \to s_2 \to \dots$ in Figure 3.5, whose second node s_2 contains r, but not q.
6. $\mathcal{M}, s_0 \vDash \mathrm{G}\,\neg(p \wedge r)$ holds since all computation paths beginning in s_0 satisfy $\mathrm{G}\,\neg(p \wedge r)$, i.e. they satisfy $\neg(p \wedge r)$ in each state along the path. Notice that $\mathrm{G}\,\phi$ holds in a state if, and only if, ϕ holds in all states reachable from the given state.
7. For similar reasons, $\mathcal{M}, s_2 \vDash \mathrm{G}\, r$ holds (note the s_2 instead of s_0).
8. For any state s of $\mathcal{M}$, we have $\mathcal{M}, s \vDash \mathrm{F}\,(\neg q \wedge r) \to \mathrm{F}\,\mathrm{G}\, r$. This says that if any path π beginning in s gets to a state satisfying $(\neg q \wedge r)$, then the path π satisfies $\mathrm{F}\,\mathrm{G}\, r$. Indeed this is true, since if the path has a state satisfying $(\neg q \wedge r)$ then (since that state must be s_2) the path does satisfy $\mathrm{F}\,\mathrm{G}\, r$. Notice what $\mathrm{F}\,\mathrm{G}\, r$ says about a path: eventually, you have continuously r.
9. The formula $\mathrm{G}\,\mathrm{F}\, p$ expresses that p occurs along the path in question infinitely often. Intuitively, it's saying: no matter how far along the path you go (that's the G part) you will find you still have a p in front of you (that's the F part). For example, the path $s_0 \to s_1 \to s_0 \to s_1 \to \dots$ satisfies $\mathrm{G}\,\mathrm{F}\, p$. But the path $s_0 \to s_2 \to s_2 \to s_2 \to \dots$ doesn't.
10. In our model, if a path from s_0 has infinitely many ps on it then it must be the path $s_0 \to s_1 \to s_0 \to s_1 \to \dots$, and in that case it also has infinitely many rs on it. So, $\mathcal{M}, s_0 \vDash \mathrm{G}\,\mathrm{F}\, p \to \mathrm{G}\,\mathrm{F}\, r$. But it is not the case the other way around! It is not the case that $\mathcal{M}, s_0 \vDash \mathrm{G}\,\mathrm{F}\, r \to \mathrm{G}\,\mathrm{F}\, p$, because we can find a path from s_0 which has infinitely many rs but only one p.

3.2.3 Practical patterns of specifications

What kind of practically relevant properties can we check with formulas of LTL? We list a few of the common patterns. Suppose atomic descriptions include some words such as busy and requested. We may require some of the following properties of real systems:

- It is impossible to get to a state where started holds, but ready does not hold: $G\neg(\mathsf{started} \wedge \neg\mathsf{ready})$
  The negation of this formula expresses that it *is possible* to get to such a state, but this is only so if interpreted on paths ($\pi \vDash \phi$). We cannot assert such a possibility if interpreted on states ($s \vDash \phi$) since we cannot express the existence of paths; for that interpretation, the negation of the formula above asserts that *all* paths will eventually get to such a state.

- For any state, if a request (of some resource) occurs, then it will eventually be acknowledged:
  $\mathrm{G}\,(\mathsf{requested} \to \mathrm{F}\,\mathsf{acknowledged})$.
- A certain process is enabled infinitely often on every computation path:
  $\mathrm{G}\,\mathrm{F}\,\mathsf{enabled}$.
- Whatever happens, a certain process will eventually be permanently deadlocked:
  $\mathrm{F}\,\mathrm{G}\,\mathsf{deadlock}$.
- If the process is enabled infinitely often, then it runs infinitely often.
  $\mathrm{G}\,\mathrm{F}\,\mathsf{enabled} \to \mathrm{G}\,\mathrm{F}\,\mathsf{running}$.
- An upwards travelling lift at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
  $\mathrm{G}\,(\mathsf{floor2} \wedge \mathsf{directionup} \wedge \mathsf{ButtonPressed5} \to (\mathsf{directionup}\ \mathrm{U}\ \mathsf{floor5}))$
  Here, our atomic descriptions are boolean expressions built from system variables, e.g., floor2.

There are some things which are *not* possible to say in LTL, however. One big class of such things are statements which assert the existence of a path, such as these ones:

- From any state *it is possible* to get to a restart state (i.e., there is a path from all states to a state satisfying restart).
- The lift *can* remain idle on the third floor with its doors closed (i.e., from the state in which it is on the third floor, there is a path along which it stays there).

LTL can't express these because it cannot directly assert the existence of paths. In Section 3.4, we look at Computation Tree Logic (CTL) which has operators for quantifying over paths, and can express these properties.

3.2.4 Important equivalences between LTL formulas

Definition 3.9 We say that two LTL formulas ϕ and ψ are semantically equivalent, or simply equivalent, writing $\phi \equiv \psi$, if for all models $\mathcal{M}$ and all paths π in $\mathcal{M}$: $\pi \vDash \phi$ iff $\pi \vDash \psi$.

The equivalence of ϕ and ψ means that ϕ and ψ are semantically interchangeable. If ϕ is a subformula of some bigger formula χ, and $\psi \equiv \phi$, then we can make the substitution of ψ for ϕ in χ without changing the meaning of χ. In propositional logic, we saw that $\wedge$ and $\vee$ are duals of each other, meaning that if you push a $\neg$ past a $\wedge$, it becomes a $\vee$, and vice versa:

$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi \qquad \neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi.$$

(Because $\wedge$ and $\vee$ are binary, pushing a negation downwards in the parse tree past one of them also has the effect of duplicating that negation.)

Similarly, F and G are duals of each other, and X is dual with itself:

$$\neg \mathrm{G}\, \phi \equiv \mathrm{F}\, \neg\phi \qquad \neg \mathrm{F}\, \phi \equiv \mathrm{G}\, \neg\phi \qquad \neg \mathrm{X}\, \phi \equiv \mathrm{X}\, \neg\phi.$$

Also U and R are duals of each other:

$$\neg(\phi \mathrel{\mathrm{U}} \psi) \equiv \neg\phi \mathrel{\mathrm{R}} \neg\psi \qquad \neg(\phi \mathrel{\mathrm{R}} \psi) \equiv \neg\phi \mathrel{\mathrm{U}} \neg\psi.$$

We should give formal proofs of these equivalences. But they are easy, so we leave them as an exercise to the reader. 'Morally' there ought to be a dual for W, and you can invent one if you like. Work out what it might mean, and then pick a symbol based on the first letter of the meaning. However, it might not be very useful.

It's also the case that F distributes over $\vee$ and G over $\wedge$, i.e.,

$$\begin{aligned} \mathrm{F}\,(\phi \vee \psi) &\equiv \mathrm{F}\, \phi \vee \mathrm{F}\, \psi \\ \mathrm{G}\,(\phi \wedge \psi) &\equiv \mathrm{G}\, \phi \wedge \mathrm{G}\, \psi. \end{aligned}$$

Compare this with the quantifier equivalences in Section 2.3.2. But F does *not* distribute over $\wedge$. What this means is that there is a model with a path which distinguishes $\mathrm{F}\,(\phi \wedge \psi)$ and $\mathrm{F}\, \phi \wedge \mathrm{F}\, \psi$, for some ϕ, ψ. Take the path $s_0 \to s_1 \to s_0 \to s_1 \to \dots$ from the system of Figure 3.3, for example; it satisfies $\mathrm{F}\, p \wedge \mathrm{F}\, r$ but it doesn't satisfy $\mathrm{F}\,(p \wedge r)$.

Here are two more equivalences in LTL:

$$\mathrm{F}\, \phi \equiv \top \mathrel{\mathrm{U}} \phi \qquad \mathrm{G}\, \phi \equiv \bot \mathrel{\mathrm{R}} \phi.$$

The first one exploits the fact that the clause for Until states two things: the second formula ϕ must become true; and until then, the first formula $\top$ must hold. So, if we put 'no constraint' for the first formula, it boils down to asking that the second formula holds, which is what F asks. (The formula $\top$ represent 'no constraint.' If you ask me to bring it about that $\top$ holds, I need do nothing, it enforces no constraint. In the same sense, $\bot$ is 'every constraint.' If you ask me to bring it about that $\bot$ holds, I'll have to meet every constraint there is, which is impossible.)

The second formula, that $\mathrm{G}\, \phi \equiv \bot \mathrel{\mathrm{R}} \phi$, can be obtained from the first by putting a $\neg$ in front of each side, and applying the duality rules. Another more intuitive way of seeing this is to recall the meaning of 'release:' $\bot$ releases ϕ, but $\bot$ will never be true, so ϕ doesn't get released.

Another pair of equivalences relates the strong and weak versions of Until, U and W. Strong until may be seen as weak until plus the constraint that the eventuality must actually occur:

$$\phi \mathrel{\mathrm{U}} \psi \equiv \phi \mathrel{\mathrm{W}} \psi \wedge \mathrm{F}\, \psi\,. \tag{3.2}$$

To prove equivalence (3.2), suppose first that a path satisfies $\phi \mathrm{U} \psi$. Then, from clause 11, we have $i \geq 1$ such that $\pi^i \vDash \psi$ and for all $j = 1, \ldots, i-1$ we have $\pi^j \vDash \phi$. From clause 12, this proves $\phi \mathrm{W} \psi$, and from clause 10 it proves $\mathrm{F} \psi$. Thus for all paths π, if $\pi \vDash \phi \mathrm{U} \psi$ then $\pi \vDash \phi \mathrm{W} \psi \wedge \mathrm{F} \psi$. As an exercise, the reader can prove it the other way around.

Writing W in terms of U is also possible: W is like U but also allows the possibility of the eventuality never occurring:

$$\phi \mathrm{W} \psi \equiv \phi \mathrm{U} \psi \vee \mathrm{G} \phi. \tag{3.3}$$

Inspection of clauses 12 and 13 reveals that R and W are rather similar. The differences are that they swap the roles of their arguments ϕ and ψ; and the clause for W has an $i - 1$ where R has i. Therefore, it is not surprising that they are expressible in terms of each other, as follows:

$$\phi \mathrm{W} \psi \equiv \psi \mathrm{R} (\phi \vee \psi) \tag{3.4}$$

$$\phi \mathrm{R} \psi \equiv \psi \mathrm{W} (\phi \wedge \psi). \tag{3.5}$$

3.2.5 Adequate sets of connectives for LTL

Recall that $\phi \equiv \psi$ holds iff any path in any transition system which satisfies ϕ also satisfies ψ, and vice versa. As in propositional logic, there is some redundancy among the connectives. For example, in Chapter 1 we saw that the set $\{\bot, \wedge, \neg\}$ forms an adequate set of connectives, since the other connectives $\vee$, $\rightarrow$, $\top$, etc., can be written in terms of those three.

Small adequate sets of connectives also exist in LTL. Here is a summary of the situation.

- X is completely orthogonal to the other connectives. That is to say, its presence doesn't help in defining any of the other ones in terms of each other. Moreover, X cannot be derived from any combination of the others.
- Each of the sets $\{\mathrm{U}, \mathrm{X}\}$, $\{\mathrm{R}, \mathrm{X}\}$, $\{\mathrm{W}, \mathrm{X}\}$ is adequate. To see this, we note that
  - R and W may be defined from U, by the duality $\phi \mathrm{R} \psi \equiv \neg(\neg\phi \mathrm{U} \neg\psi)$ and equivalence (3.4) followed by the duality, respectively.
  - U and W may be defined from R, by the duality $\phi \mathrm{U} \psi \equiv \neg(\neg\phi \mathrm{R} \neg\psi)$ and equivalence (3.4), respectively.
  - R and U may be defined from W, by equivalence (3.5) and the duality $\phi \mathrm{U} \psi \equiv \neg(\neg\phi \mathrm{R} \neg\psi)$ followed by equivalence (3.5).

Sometimes it is useful to look at adequate sets of connectives which do not rely on the availability of negation. That's because it is often convenient to assume formulas are written in negation-normal form, where all the negation symbols are applied to propositional atoms (i.e., they are near the leaves of the parse tree). In this case, these sets are adequate for the fragment without X, and no strict subset is: $\{\mathrm{U}, \mathrm{R}\}$, $\{\mathrm{U}, \mathrm{W}\}$, $\{\mathrm{U}, \mathrm{G}\}$, $\{\mathrm{R}, \mathrm{F}\}$, $\{\mathrm{W}, \mathrm{F}\}$. But $\{\mathrm{R}, \mathrm{G}\}$ and $\{\mathrm{W}, \mathrm{G}\}$ are not adequate. Note that one cannot define G with {U,F}, and one cannot define F with {R,G} or {W,G}.

We finally state and prove a useful equivalence about U.

Theorem 3.10 The equivalence $\phi \mathrm{~U~} \psi \ \equiv\ \neg(\neg\psi \mathrm{~U~} (\neg\phi \land \neg\psi)) \land \mathrm{F}\,\psi$ holds for all LTL formulas ϕ and ψ.

PROOF: Take any path $s_0 \to s_1 \to s_2 \to \dots$ in any model.

First, suppose $s_0 \vDash \phi \mathrm{~U~} \psi$ holds. Let n be the smallest number such that $s_n \vDash \psi$; such a number has to exist since $s_0 \vDash \phi \mathrm{~U~} \psi$; then, for each $k < n$, $s_k \vDash \phi$. We immediately have $s_0 \vDash \mathrm{F}\,\psi$, so it remains to show $s_0 \vDash \neg(\neg\psi \mathrm{~U~} (\neg\phi \land \neg\psi))$, which, if we expand, means:
$(*)$ for each $i > 0$, if $s_i \vDash \neg\phi \land \neg\psi$, then there is some $j < i$ with $s_j \vDash \psi$.
Take any $i > 0$ with $s_i \vDash \neg\phi \land \neg\psi$; $i > n$, so we can take $j \overset{\mathrm{def}}{=} n$ and have $s_j \vDash \psi$.
Conversely, suppose $s_0 \vDash \neg(\neg\psi \mathrm{~U~} (\neg\phi \land \neg\psi)) \land \mathrm{F}\,\psi$ holds; we prove $s_0 \vDash \phi \mathrm{~U~} \psi$. Since $s_0 \vDash \mathrm{F}\,\psi$, we have a minimal n as before. We show that, for any $i < n$, $s_i \vDash \phi$. Suppose $s_i \vDash \neg\phi$; since n is minimal, we know $s_i \vDash \neg\psi$, so by $(*)$ there is some $j < i < n$ with $s_j \vDash \psi$, contradicting the minimality of n. □

3.3 Model checking: systems, tools, properties

3.3.1 Example: mutual exclusion

Let us now look at a larger example of verification using LTL, having to do with *mutual exclusion*. When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time. Several processes simultaneously editing the same file would not be desirable.

We therefore identify certain *critical sections* of each process' code and arrange that only one process can be in its critical section at a time. The critical section should include all the access to the shared resource (though it should be as small as possible so that no unnecessary exclusion takes place). The problem we are faced with is to find a *protocol* for determining which process is allowed to enter its critical section at which time. Once we have found one which we think works, we verify our solution by checking that it has some expected properties, such as the following ones:

Safety: Only one process is in its critical section at any time.

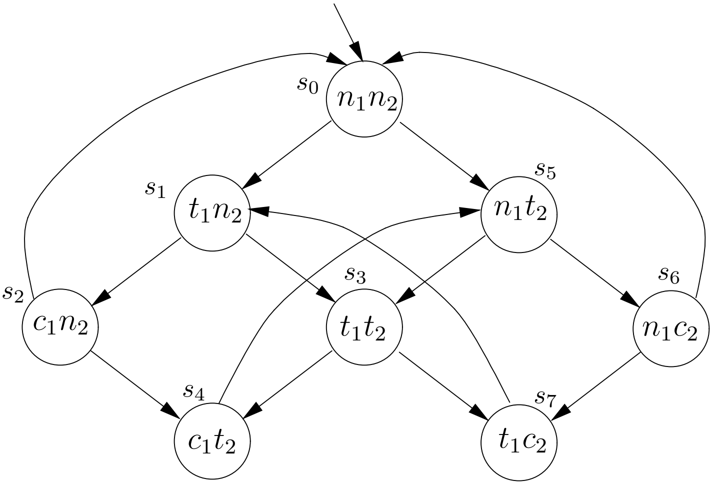


Figure 3.7. A first-attempt model for mutual exclusion.

This safety property is not enough, since a protocol which permanently excluded every process from its critical section would be safe, but not very useful. Therefore, we should also require:

Liveness: Whenever any process requests to enter its critical section, it will eventually be permitted to do so.

Non-blocking: A process can always request to enter its critical section.

Some rather crude protocols might work on the basis that they cycle through the processes, making each one in turn enter its critical section. Since it might be naturally the case that some of them request access to the shared resource more often than others, we should make sure our protocol has the property:

No strict sequencing: Processes need not enter their critical section in strict sequence.

The first modelling attempt We will model two processes, each of which is in its non-critical state (n), or trying to enter its critical state (t), or in its critical state (c). Each individual process undergoes transitions in the cycle $n \to t \to c \to n \to \dots$, but the two processes interleave with each other. Consider the protocol given by the transition system $\mathcal{M}$ in Figure 3.7. (As usual, we write $p_1p_2 \dots p_m$ in a node s to denote that $p_1, p_2, \dots, p_m$ are the only propositional atoms true at s.) The two processes start off in their non-critical sections (global state s_0). State s_0 is the only *initial* state, indicated by the incoming edge with no source. Either of them may now move to its trying state, but only one of them can ever make a transition at a time (*asynchronous interleaving*). At each step, an (unspecified) scheduler determines which process may run. So there is a transition arrow from s_0 to s_1 and s_5. From s_1 (i.e., process 1 trying, process 2 non-critical) again two things can happen: either process 1 moves again (we go to s_2), or process 2 moves (we go to s_3). Notice that not every process can move in every state. For example, process 1 cannot move in state s_7, since it cannot go into its critical section until process 2 comes out of its critical section.

We would like to check the four properties by first describing them as temporal logic formulas. Unfortunately, they are not all expressible as LTL formulas. Let us look at them case-by-case.

Safety: This is expressible in LTL, as $\mathrm{G}\,\neg(c_1 \wedge c_2)$. Clearly, $\mathrm{G}\,\neg(c_1 \wedge c_2)$ is satisfied in the initial state (indeed, in every state).

Liveness: This is also expressible: $\mathrm{G}\,(t_1 \to \mathrm{F}\,c_1)$. However, it is *not* satisfied by the initial state, for we can find a path starting at the initial state along which there is a state, namely s_1, in which t_1 is true but from there along the path c_1 is false. The path in question is $s_0 \to s_1 \to s_3 \to s_7 \to s_1 \to s_3 \to s_7 \dots$ on which c_1 is always false.

Non-blocking: Let's just consider process 1. We would like to express the property as: for every state satisfying n_1, there is a successor satisfying t_1. Unfortunately, this existence quantifier on paths ('there is a successor satisfying...') cannot be expressed in LTL. It can be expressed in the logic CTL, which we will turn to in the next section (for the impatient, see page 215).

No strict sequencing: We might consider expressing this as saying: there is a path with two distinct states satisfying c_1 such that no state in between them has that property. However, we cannot express 'there exists a path,' so let us consider the complement formula instead. The complement says that all paths having a c_1 period which ends cannot have a further c_1 state until a c_2 state occurs. We write this as: $\mathrm{G}\,(c_1 \to c_1\ \mathrm{W}\ (\neg c_1 \wedge \neg c_1\ \mathrm{W}\ c_2))$. This says that anytime we get into a c_1 state, either that condition persists indefinitely, or it ends with a non-c_1 state and in that case there is no further c_1 state unless and until we obtain a c_2 state.

This formula is false, as exemplified by the path $s_0 \to s_5 \to s_3 \to s_4 \to s_5 \to s_3 \to s_4 \dots$. Therefore the original condition expressing that strict sequencing need not occur, is true.

Before further considering the mutual exclusion example, some comments about expressing properties in LTL are appropriate. Notice that in the no-strict-sequencing property, we overcame the problem of not being able to express the existence of paths by instead expressing the complement property, which of course talks about all paths. Then we can perform our check, and simply reverse the answer; if the complement property is false, we declare our property to be true, and vice versa.

Why was that tactic not available to us to express the non-blocking property? The reason is that it says: every path to a n_1 state may be continued by a one-step path to a t_1 state. The presence of both universal and existential quantifiers is the problem. In the no-strict-sequencing property, we had only an existential quantifier; thus, taking the complement property turned it into a universal path quantifier, which can be expressed in LTL. But where we have alternating quantifiers, taking the complement property doesn't help in general.

Let's go back to the mutual exclusion example. The reason liveness failed in our first attempt at modelling mutual exclusion is that non-determinism means it *might* continually favour one process over another. The problem is that the state s_3 does not distinguish between which of the processes *first* went into its trying state. We can solve this by splitting s_3 into two states.

The second modelling attempt The two states s_3 and s_9 in Figure 3.8 both correspond to the state s_3 in our first modelling attempt. They both record that the two processes are in their trying states, but in s_3 it is implicitly recorded that it is process 1's turn, whereas in s_9 it is process 2's turn. Note that states s_3 and s_9 both have the labelling $t_1 t_2$; the definition of transition systems does not preclude this. We can think of there being some other, hidden, variables which are not part of the initial labelling, which distinguish s_3 and s_9.

Remark 3.11 The four properties of safety, liveness, non-blocking and no-strict-sequencing are satisfied by the model in Figure 3.8. (Since the non-blocking property has not yet been written in temporal logic, we can only check it informally.)

In this second modelling attempt, our transition system is still slightly over-simplified, because we are assuming that it will move to a different state on every tick of the clock (there are no transitions to the same state). We may wish to model that a process can stay in its critical state for several ticks, but if we include an arrow from s_4, or s_7, to itself, we will again violate liveness. This problem will be solved later in this chapter when we consider 'fairness constraints' (Section 3.6.2).

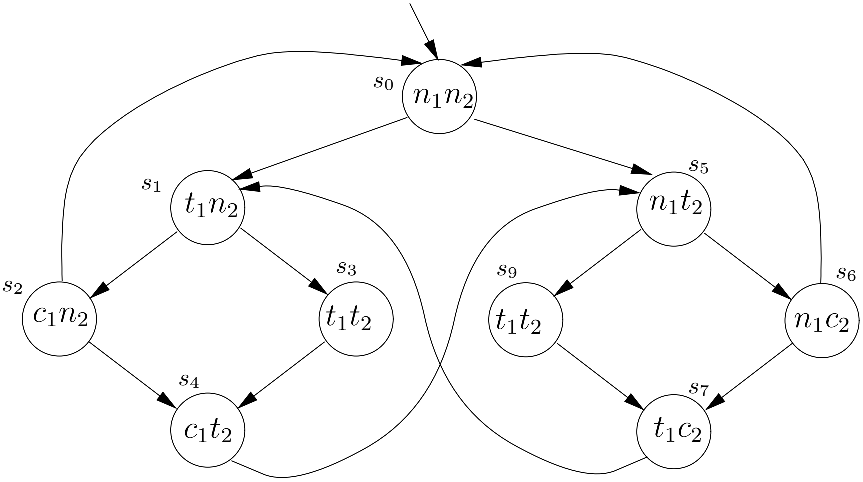


Figure 3.8. A second-attempt model for mutual exclusion. There are now two states representing t_1t_2, namely s_3 and s_9.

3.3.2 The NuSMV model checker

So far, this chapter has been quite theoretical; and the sections after this one continue in this vein. However, one of the exciting things about model checking is that it is also a practical subject, for there are several efficient implementations which can check large systems in realistic time. In this section, we look at the NuSMV model-checking system. NuSMV stands for 'New Symbolic Model Verifier.' NuSMV is an Open Source product, is actively supported and has a substantial user community. For details on how to obtain it, see the bibliographic notes at the end of the chapter.

NuSMV (sometimes called simply SMV) provides a language for describing the models we have been drawing as diagrams and it directly checks the validity of LTL (and also CTL) formulas on those models. SMV takes as input a text consisting of a program describing a model and some specifications (temporal logic formulas). It produces as output either the word 'true' if the specifications hold, or a trace showing why the specification is false for the model represented by our program.

SMV programs consist of one or more modules. As in the programming language C, or Java, one of the modules must be called `main`. Modules can declare variables and assign to them. Assignments usually give the `init`ial value of a variable and its `next` value as an expression in terms of the current values of variables. This expression can be non-deterministic (denoted by several expressions in braces, or no assignment at all). Non-determinism is used to model the environment and for abstraction.

The following input to SMV:

```
MODULE main
VAR
  request : boolean;
  status : {ready,busy};
ASSIGN
  init(status) := ready;
  next(status) := case
                    request : busy;
                    1 : {ready,busy};
                  esac;
LTLSPEC
  G(request -> F status=busy)
```

consists of a program and a specification. The program has two variables, `request` of type `boolean` and `status` of enumeration type {`ready, busy`}: `0` denotes 'false' and `1` represents 'true.' The initial and subsequent values of variable `request` are not determined within this program; this conservatively models that these values are determined by an external environment. This under-specification of `request` implies that the value of variable `status` is partially determined: initially, it is ready; and it becomes busy whenever `request` is true. If `request` is false, the next value of `status` is not determined.

Note that the case `1:` signifies the default case, and that case statements are evaluated from the top down: if several expressions to the left of a ':' are true, then the command corresponding to the first, top-most true expression will be executed. The program therefore denotes the transition system shown in Figure 3.9; there are four states, each one corresponding to a possible value of the two binary variables. Note that we wrote 'busy' as a shorthand for '`status`=`busy`' and 'req' for '`request` is true.'

It takes a while to get used to the syntax of SMV and its meaning. Since variable `request` functions as a genuine environment in this model, the program and the transition system are *non-deterministic*: i.e., the 'next state' is not uniquely defined. Any state transition based on the behaviour of `status` comes in a pair: to a successor state where `request` is false, or true, respectively. For example, the state '$\neg$req, busy' has four states it can move to (itself and three others).

LTL specifications are introduced by the keyword `LTLSPEC` and are simply LTL formulas. Notice that SMV uses &, |, `->` and `!` for $\wedge$, $\vee$, $\rightarrow$ and $\neg$, respectively, since they are available on standard keyboards. We may

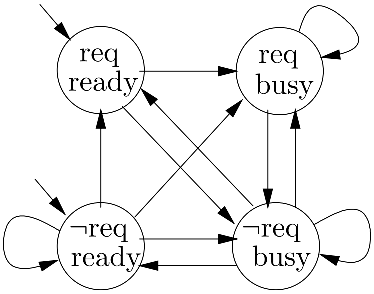


Figure 3.9. The model corresponding to the SMV program in the text.

easily verify that the specification of our module `main` holds of the model in Figure 3.9.

Modules in SMV SMV supports breaking a system description into several *modules*, to aid readability and to verify interaction properties. A module is instantiated when a variable having that module name as its type is declared. This defines a set of variables, one for each one declared in the module description. In the example below, which is one of the ones distributed with SMV, a counter which repeatedly counts from 000 through to 111 is described by three single-bit counters. The module `counter_cell` is instantiated three times, with the names `bit0`, `bit1` and `bit2`. The counter module has one formal parameter, `carry_in`, which is given the actual value 1 in `bit0`, and `bit0.carry_out` in the instance `bit1`. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`. Note that we use the period '.' in `m.v` to access the variable `v` in module `m`. This notation is also used by Alloy (see Chapter 2) and a host of programming languages to access fields in record structures, or methods in objects. The keyword `DEFINE` is used to assign the expression `value & carry_in` to the symbol `carry_out` (such definitions are just a means for referring to the current value of a certain expression).

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
LTLSPEC
  G F bit2.carry_out
```

```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := (value + carry_in) mod 2;
DEFINE
  carry_out := value & carry_in;
```

The effect of the DEFINE statement could have been obtained by declaring a new variable and assigning its value thus:

```
VAR
  carry_out : boolean;
ASSIGN
  carry_out := value & carry_in;
```

Notice that, in this assignment, the *current* value of the variable is assigned. Defined symbols are usually preferable to variables, since they don't increase the state space by declaring new variables. However, they cannot be assigned non-deterministically since they refer only to another expression.

Synchronous and asynchronous composition By default, modules in SMV are composed *synchronously*: this means that there is a global clock and, each time it ticks, each of the modules executes in parallel. By use of the `process` keyword, it is possible to compose the modules asynchronously. In that case, they run at different 'speeds,' interleaving arbitrarily. At each tick of the clock, *one* of them is non-deterministically chosen and executed for one cycle. Asynchronous interleaving composition is useful for describing communication protocols, asynchronous circuits and other systems whose actions are not synchronised to a global clock.

The bit counter above is synchronous, whereas the examples below of mutual exclusion and the alternating bit protocol are asynchronous.

3.3.3 Running NuSMV

The normal use of NuSMV is to run it in batch mode, from a Unix shell or command prompt in Windows. The command line

```
NuSMV counter3.smv
```

will analyse the code in the file `counter3.smv` and report on the specifications it contains. One can also run NuSMV interactively. In that case, the command line

```
NuSMV -int counter3.smv
```

enters NuSMV's command-line interpreter. From there, there is a variety of commands you can use which allow you to compile the description and run the specification checks, as well as inspect partial results and set various parameters. See the NuSMV user manual for more details.

NuSMV also supports *bounded model checking*, invoked by the command-line option `-bmc`. Bounded model checking looks for counterexamples in order of size, starting with counterexamples of length 1, then 2, etc., up to a given threshold (10 by default). Note that bounded model checking is incomplete: failure to find a counterexample does not mean that there is none, but only that there is none of length up to the threshold. For related reasons, this incompleteness features also in Alloy and its constraint analyzer. Thus, while a negative answer can be relied on (if NuSMV finds a counterexample, it is valid), a positive one cannot. References on bounded model checking can be found in the bibliographic notes on page 254. Later on, we use bounded model checking to prove the optimality of a scheduler.

3.3.4 Mutual exclusion revisited

Figure 3.10 gives the SMV code for a mutual exclusion protocol. This code consists of two modules, `main` and `prc`. The module `main` has the variable `turn`, which determines whose turn it is to enter the critical section if both are trying to enter (recall the discussion about the states s_3 and s_9 in Section 3.3.1).

The module `main` also has two instantiations of `prc`. In each of these instantiations, `st` is the status of a process (saying whether it is in its critical section, or not, or trying) and `other-st` is the status of the other process (notice how this is passed as a parameter in the third and fourth lines of `main`).

The value of `st` evolves in the way described in a previous section: when it is n, it may stay as n or move to t. When it is t, if the other one is n, it will go straight to c, but if the other one is t, it will check whose turn it is before going to c. Then, when it is c, it may move back to n. Each instantiation of `prc` gives the turn to the other one when it gets to its critical section.

An important feature of SMV is that we can restrict its search tree to execution paths along which an arbitrary boolean formula about the state

```
MODULE main
   VAR
      pr1: process prc(pr2.st, turn, 0);
      pr2: process prc(pr1.st, turn, 1);
      turn: boolean;
   ASSIGN
      init(turn) := 0;
   -- safety
   LTLSPEC  G!((pr1.st = c) & (pr2.st = c))
   -- liveness
   LTLSPEC  G((pr1.st = t) -> F (pr1.st = c))
   LTLSPEC  G((pr2.st = t) -> F (pr2.st = c))
   -- ‘negation’ of strict sequencing (desired to be false)
   LTLSPEC G(pr1.st=c -> ( G pr1.st=c | (pr1.st=c U
             (!pr1.st=c & G !pr1.st=c | ((!pr1.st=c) U pr2.st=c)))))

MODULE prc(other-st, turn, myturn)
   VAR
      st: {n, t, c};
   ASSIGN
      init(st) := n;
      next(st) :=
         case
            (st = n)                                  : {t,n};
            (st = t) & (other-st = n)                 : c;
            (st = t) & (other-st = t) & (turn = myturn): c;
            (st = c)                                  : {c,n};
            1                                         : st;
         esac;
      next(turn) :=
         case
            turn = myturn & st = c : !turn;
            1                      : turn;
         esac;
   FAIRNESS running
   FAIRNESS  !(st = c)
```

Figure 3.10. SMV code for mutual exclusion. Because W is not supported by SMV, we had to make use of equivalence (3.3) to write the no-strict-sequencing formula as an equivalent but longer formula involving U.

ϕ is true infinitely often. Because this is often used to model *fair* access to resources, it is called a *fairness constraint* and introduced by the keyword `FAIRNESS`. Thus, the occurrence of `FAIRNESS` ϕ means that SMV, when checking a specification ψ, will ignore any path along which ϕ is not satisfied infinitely often.

In the module `prc`, we restrict model checks to computation paths along which `st` is infinitely often not equal to c. This is because our code allows the process to stay in its critical section as long as it likes. Thus, there is another opportunity for liveness to fail: if process 2 stays in its critical section forever, process 1 will never be able to enter. Again, we ought not to take this kind of violation into account, since it is patently unfair if a process is allowed to stay in its critical section for ever. We are looking for more subtle violations of the specifications, if there are any. To avoid the one above, we stipulate the fairness constraint `!(st=c)`.

If the module in question has been declared with the `process` keyword, then at each time point SMV will non-deterministically decide whether or not to select it for execution, as explained earlier. We may wish to ignore paths in which a module is starved of processor time. The reserved word `running` can be used instead of a formula in a fairness constraint: writing `FAIRNESS running` restricts attention to execution paths along which the module in which it appears is selected for execution infinitely often.

In `prc`, we restrict ourselves to such paths, since, without this restriction, it would be easy to violate the liveness constraint if an instance of `prc` were *never* selected for execution. We assume the scheduler is fair; this assumption is codified by two `FAIRNESS` clauses. We return to the issue of fairness, and the question of how our model-checking algorithm copes with it, in the next section.

Please run this program in NuSMV to see which specifications hold for it.

The transition system corresponding to this program is shown in Figure 3.11. Each state shows the values of the variables; for example, ct1 is the state in which process 1 and 2 are critical and trying, respectively, and turn=1. The labels on the transitions show which process was selected for execution. In general, each state has several transitions, some in which process 1 moves and others in which process 2 moves.

This model is a bit different from the previous model given for mutual exclusion in Figure 3.8, for these two reasons:

- Because the boolean variable `turn` has been explicitly introduced to distinguish between states s_3 and s_9 of Figure 3.8, we now distinguish between certain states

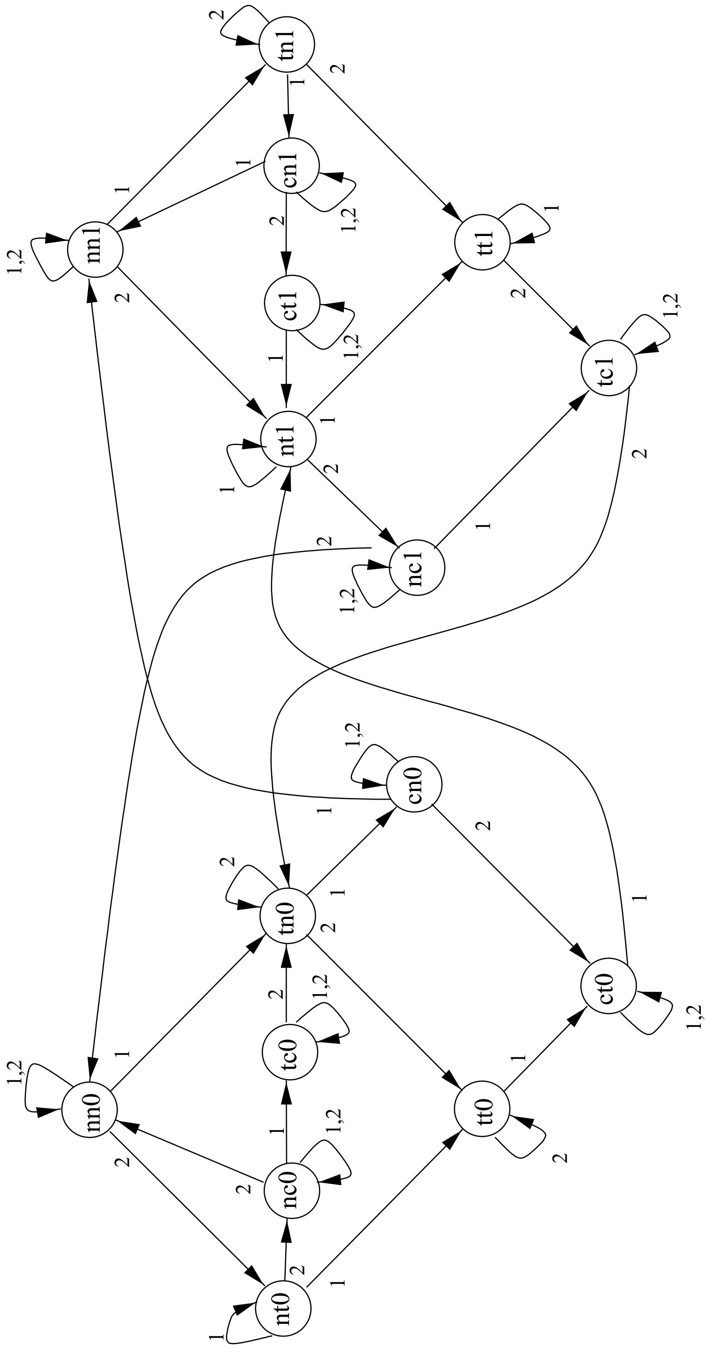


Figure 3.11. The transition system corresponding to the SMV code in Figure 3.10. The labels on the transitions denote the process which makes the move. The label $1, 2$ means that either process could make that move.

(for example, ct0 and ct1) which were identical before. However, these states are not distinguished if you look just at the transitions *from* them. Therefore, they satisfy the same LTL formulas which don't mention `turn`. Those states are distinguished only by the way they can arise.

- We have eliminated an over-simplification made in the model of Figure 3.8. Recall that we assumed the system would move to a different state on every tick of the clock (there were no transitions from a state to itself). In Figure 3.11, we allow transitions from each state to itself, representing that a process was chosen for execution and did some private computation, but did not move in or out of its critical section. Of course, by doing this we have introduced paths in which one process gets stuck in its critical section, whence the need to invoke a fairness constraint to eliminate such paths.

3.3.5 The ferryman

You may recall the puzzle of a ferryman, goat, cabbage, and wolf all on one side of a river. The ferryman can cross the river with at most one passenger in his boat. There is a behavioural conflict between:

1. the goat and the cabbage; and
2. the goat and the wolf;

if they are on the same river bank but the ferryman crosses the river or stays on the other bank.

Can the ferryman transport all goods to the other side, without any conflicts occurring? This is a *planning problem,* but it can be solved by model checking. We describe a transition system in which the states represent which goods are at which side of the river. Then we ask if the goal state is reachable from the initial state: Is there a path from the initial state such that it has a state along it at which all the goods are on the other side, and during the transitions to that state the goods are never left in an unsafe, conflicting situation?

We model all possible behaviour (including that which results in conflicts) as a NuSMV program (Figure 3.12). The location of each agent is modelled as a boolean variable: 0 denotes that the agent is on the initial bank, and 1 the destination bank. Thus, `ferryman` $= 0$ means that the ferryman is on the initial bank, `ferryman` $= 1$ that he is on the destination bank, and similarly for the variables `goat`, `cabbage` and `wolf`.

The variable `carry` takes a value indicating whether the goat, cabbage, wolf or nothing is carried by the ferryman. The definition of `next(carry)` works as follows. It is non-deterministic, but the set from which a value is non-deterministically chosen is determined by the values of ferryman, goat,

```
MODULE main
 VAR
  ferryman : boolean;
  goat     : boolean;
  cabbage  : boolean;
  wolf     : boolean;
  carry    : {g,c,w,0};
ASSIGN
 init(ferryman) := 0; init(goat)     := 0;
 init(cabbage)  := 0; init(wolf)     := 0;
 init(carry)    := 0;

 next(ferryman) := 0,1;

 next(carry) := case
                  ferryman=goat : g;
                  1             : 0;
               esac union
               case
                  ferryman=cabbage : c;
                  1                : 0;
               esac union
               case
                  ferryman=wolf : w;
                  1             : 0;
               esac union 0;

 next(goat) := case
   ferryman=goat  & next(carry)=g : next(ferryman);
   1                              : goat;
 esac;
 next(cabbage) := case
   ferryman=cabbage & next(carry)=c : next(ferryman);
   1                                : cabbage;
 esac;
 next(wolf) := case
   ferryman=wolf & next(carry)=w : next(ferryman);
   1                             : wolf;
 esac;

LTLSPEC !((   (goat=cabbage | goat=wolf) -> goat=ferryman)
            U (cabbage & goat & wolf & ferryman))
```

Figure 3.12. NuSMV code for the ferryman planning problem.

etc., and always includes 0. If `ferryman` = `goat` (i.e., they are on the same side) then `g` is a member of the set from which `next(carry)` is chosen. The situation for cabbage and wolf is similar. Thus, if `ferryman` = `goat` = `wolf` ≠ `cabbage` then that set is {`g`, `w`, `0`}. The next value assigned to `ferryman` is non-deterministic: he can choose to cross or not to cross the river. But the next values of `goat`, `cabbage` and `wolf` are deterministic, since whether they are carried or not is determined by the ferryman's choice, represented by the non-deterministic assignment to `carry`; these values follow the same pattern.

Note how the boolean guards refer to state bits at the next state. The SMV compiler does a dependency analysis and rejects circular dependencies on next values. (The dependency analysis is rather pessimistic: sometimes NuSMV complains of circularity even in situations when it could be resolved. The original CMU-SMV is more liberal in this respect.)

Running NuSMV We seek a path satisfying $\phi \mathrm{~U~} \psi$, where ψ asserts the final goal state, and ϕ expresses the safety condition (if the goat is with the cabbage or the wolf, then the ferryman is there, too, to prevent any untoward behaviour). Thus, we assert that all paths satisfy $\neg(\phi \mathrm{~U~} \psi)$, i.e., no path satisfies $\phi \mathrm{~U~} \psi$. We hope this is not the case, and NuSMV will give us an example path which does satisfy $\phi \mathrm{~U~} \psi$. Indeed, running NuSMV gives us the path of Figure 3.13, which represents a solution to the puzzle.

The beginning of the generated path represents the usual solution to this puzzle: the ferryman takes the goat first, then goes back for the cabbage. To avoid leaving the goat and the cabbage together, he takes the goat back, and picks up the wolf. Now the wolf and the cabbage are on the destination side, and he goes back again to get the goat. This brings us to State 1.9, where the ferryman appears to take a well-earned break. But the path continues. States 1.10 to 1.15 show that he takes his charges back to the original side of the bank; first the cabbage, then the wolf, then the goat. Unfortunately it appears that the ferryman's clever plan up to state 1.9 is now spoiled, because the goat meets an unhappy end in state 1.11.

What went wrong? Nothing, actually. NuSMV has given us an infinite path, which loops around the 15 illustrated states. Along the infinite path, the ferryman repeatedly takes his goods across (safely), and then back again (unsafely). This path does indeed satisfy the specification $\phi \mathrm{~U~} \psi$, which asserts the safety of the forward journey but says nothing about what happens after that. In other words, the path is correct; it satisfies $\phi \mathrm{~U~} \psi$ (with ψ occurring at state 8). What happens along the path after that has no bearing on $\phi \mathrm{~U~} \psi$.

```
acws-0116% nusmv  ferryman.smv
*** This is NuSMV 2.1.2 (compiled 2002-11-22 12:00:00)
*** For more information of NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv-users@irst.itc.it>.
-- specification !(((goat = cabbage | goat = wolf) -> goat = ferryman)
                 U (((cabbage & goat) & wolf) & ferryman)) is false
-- as demonstrated by the following execution sequence
-- loop starts here --
-> State 1.1 <-
    ferryman = 0              -> State 1.8 <-
    goat = 0                      ferryman = 1
    cabbage = 0                   goat = 1
    wolf = 0                      carry = g
    carry = 0                 -> State 1.9 <-
-> State 1.2 <-               -> State 1.10 <-
    ferryman = 1                  ferryman = 0
    goat = 1                      cabbage = 0
    carry = g                     carry = c
-> State 1.3 <-               -> State 1.11 <-
    ferryman = 0                  ferryman = 1
    carry = 0                     carry = 0
-> State 1.4 <-               -> State 1.12 <-
    ferryman = 1                  ferryman = 0
    cabbage = 1                   wolf = 0
    carry = c                     carry = w
-> State 1.5 <-               -> State 1.13 <-
    ferryman = 0                  ferryman = 1
    goat = 0                      carry = 0
    carry = g                 -> State 1.14 <-
-> State 1.6 <-                   ferryman = 0
    ferryman = 1                  goat = 0
    wolf = 1                      carry = g
    carry = w                 -> State 1.15 <-
-> State 1.7 <-                   carry = 0
    ferryman = 0
    carry = 0
```

Figure 3.13. A solution path to the ferryman puzzle. It is unnecessarily long. Using bounded model checking will refine it into an optimal solution.

Invoking *bounded model checking* will produce the shortest possible path to violate the property; in this case, it is states 1.1 to 1.8 of the illustrated path. It is the shortest, *optimal* solution to our planning problem since the model check `NuSMV -bmc -bmc_length 7 ferryman.smv` shows that the LTL formula holds in that model, meaning that no solution with fewer than seven transitions is possible.

One might wish to verify whether there is a solution which involves three journeys for the goat. This can be done by altering the LTL formula. Instead of seeking a path satisfying ϕ U ψ, where ϕ equals $(\texttt{goat} = \texttt{cabbage} \lor \texttt{goat} = \texttt{wolf}) \to \texttt{goat} = \texttt{ferryman}$ and ψ equals $\texttt{cabbage} \land \texttt{goat} \land \texttt{wolf} \land \texttt{ferryman}$, we now seek a path satisfying $(\phi \text{ U } \psi) \land \text{G}\,(\texttt{goat} \to \text{G}\,\texttt{goat})$. The last bit says that once the goat has crossed, he remains across; otherwise, the goat makes at least three trips. NuSMV verifies that the negation of this formula is true, confirming that there is no such solution.

3.3.6 The alternating bit protocol

The alternating bit protocol (ABP) is a protocol for transmitting messages along a 'lossy line,' i.e., a line which may lose or duplicate messages. The protocol guarantees that, providing the line doesn't lose infinitely many messages, communication between the sender and the receiver will be successful. (We allow the line to lose or duplicate messages, but it may not corrupt messages; however, there is no way of guaranteeing successful transmission along a line which can corrupt.)

The ABP works as follows. There are four entities, or agents: the sender, the receiver, the message channel and the acknowledgement channel. The sender transmits the first part of the message together with the 'control' bit 0. If, and when, the receiver receives a message with the control bit 0, it sends 0 along the acknowledgement channel. When the sender receives this acknowledgement, it sends the next packet with the control bit 1. If and when the receiver receives this, it acknowledges by sending a 1 on the acknowledgement channel. By alternating the control bit, both receiver and sender can guard against duplicating messages and losing messages (i.e., they ignore messages that have the unexpected control bit).

If the sender doesn't get the expected acknowledgement, it continually resends the message, until the acknowledgement arrives. If the receiver doesn't get a message with the expected control bit, it continually resends the previous acknowledgement.

Fairness is also important for the ABP. It comes in because, although we want to model the fact that the channel can lose messages, we want to assume that, if we send a message often enough, eventually it will arrive. In other words, the channel cannot lose an infinite sequence of messages. If we did not make this assumption, then the channels could lose all messages and, in that case, the ABP would not work.

Let us see this in the concrete setting of SMV. We may assume that the text to be sent is divided up into single-bit messages, which are sent

```
MODULE sender(ack)
VAR
   st       : {sending,sent};
   message1 : boolean;
   message2 : boolean;
ASSIGN
   init(st) := sending;
   next(st) := case
                  ack = message2 & !(st=sent) : sent;
                  1                 : sending;
               esac;
   next(message1) :=
               case
                  st = sent : {0,1};
                  1         : message1;
               esac;
   next(message2) :=
               case
                  st = sent : !message2;
                  1         : message2;
               esac;
FAIRNESS running
LTLSPEC G F st=sent
```

Figure 3.14. The ABP sender in SMV.

sequentially. The variable message1 is the current bit of the message being sent, whereas message2 is the control bit. The definition of the module sender is given in Figure 3.14. This module spends most of its time in st=sending, going only briefly to st=sent when it receives an acknowledgement corresponding to the control bit of the message it has been sending. The variables message1 and message2 represent the actual data being sent and the control bit, respectively. On successful transmission, the module obtains a new message to send and returns to st=sending. The new message1 is obtained non-deterministically (i.e., from the environment); message2 alternates in value. We impose FAIRNESS running, i.e., the sender must be selected to run infinitely often. The LTLSPEC tests that we can always succeed in sending the current message. The module receiver is programmed in a similar way, in Figure 3.15.

We also need to describe the two channels, in Figure 3.16. The acknowledgement channel is an instance of the one-bit channel one-bit-chan below. Its lossy character is specified by the assignment to forget. The value of

```
MODULE receiver(message1,message2)
VAR
   st       : {receiving,received};
   ack      : boolean;
   expected : boolean;
ASSIGN
   init(st) := receiving;
   next(st) := case
                  message2=expected & !(st=received) : received;
                  1                                  : receiving;
               esac;
   next(ack) :=
               case
                  st = received : message2;
                  1             : ack;
               esac;
   next(expected) :=
               case
                  st = received : !expected;
                  1             : expected;
               esac;
FAIRNESS running
LTLSPEC G F st=received
```

Figure 3.15. The ABP receiver in SMV.

input should be transmitted to output, unless forget is true. The two-bit channel two-bit-chan, used to send messages, is similar. Again, the non-deterministic variable forget determines whether the current bit is lost or not. Either both parts of the message get through, or neither of them does (the channel is assumed not to corrupt messages).

The channels have fairness constraint which are intended to model the fact that, although channels can lose messages, we assume that they infinitely often transmit the message correctly. (If this were not the case, then we could find an uninteresting violation of the liveness property, for example a path along which all messages from a certain time onwards get lost.)

It is interesting to note that the fairness constraint 'infinitely often !forget' is not sufficient to prove the desired properties, for although it forces the channel to transmit infinitely often, it doesn't prevent it from (say) dropping all the 0 bits and transmitting all the 1 bits. That is why we use the stronger fairness constraints shown. Some systems allow fairness

```
MODULE one-bit-chan(input)
VAR
   output : boolean;
   forget : boolean;
ASSIGN
   next(output) := case
                      forget : output;
                      1:       input;
                    esac;
FAIRNESS running
FAIRNESS input & !forget
FAIRNESS !input & !forget

MODULE two-bit-chan(input1,input2)
VAR
   forget : boolean;
   output1 : boolean;
   output2 : boolean;
ASSIGN
   next(output1) := case
                      forget : output1;
                      1:       input1;
                    esac;
   next(output2) := case
                      forget : output2;
                      1:       input2;
                    esac;
FAIRNESS running
FAIRNESS input1 & !forget
FAIRNESS !input1 & !forget
FAIRNESS input2 & !forget
FAIRNESS !input2 & !forget
```

Figure 3.16. The two modules for the two ABP channels in SMV.

contraints of the form ‘infinitely often p implies infinitely often q’, which would be more satisfactory here, but is not allowed by SMV.

Finally, we tie it all together with the module `main` (Figure 3.17). Its role is to connect together the components of the system, and giving them initial values of their parameters. Since the first control bit is 0, we also initialise the receiver to expect a 0. The receiver should start off by sending 1 as its

```
MODULE main
VAR
   s : process sender(ack_chan.output);
   r : process receiver(msg_chan.output1,msg_chan.output2);
   msg_chan : process two-bit-chan(s.message1,s.message2);
   ack_chan : process one-bit-chan(r.ack);
ASSIGN
   init(s.message2) := 0;
   init(r.expected) := 0;
   init(r.ack)      := 1;
   init(msg_chan.output2) := 1;
   init(ack_chan.output) := 1;

LTLSPEC  G (s.st=sent & s.message1=1 -> msg_chan.output1=1)
```

Figure 3.17. The `main` ABP module.

acknowledgement, so that `sender` does not think that its very first message is being acknowledged before anything has happened. For the same reason, the output of the channels is initialised to 1.

The specifications for ABP. Our SMV program satisfies the following specifications:

Safety: If the message bit 1 has been sent and the correct acknowledgement has been returned, then a 1 was indeed received by the receiver: `G (S.st=sent & S.message1=1 -> msg_chan.output1=1)`.

Liveness: Messages get through eventually. Thus, for any state there is inevitably a future state in which the current message has got through. In the module `sender`, we specified `G F st=sent`. (This specification could equivalently have been written in the main module, as `G F S.st=sent`.) Similarly, acknowledgements get through eventually. In the module `receiver`, we write `G F st=received`.

3.4 Branching-time logic

In our analysis of LTL (*linear-time temporal logic*) in the preceding sections, we noted that LTL formulas are evaluated on *paths*. We defined that a *state* of a system satisfies an LTL formula if *all paths* from the given state satisfy it. Thus, LTL implicitly quantifies universally over paths. Therefore, properties which assert the existence of a path cannot be expressed in LTL. This problem can partly be alleviated by considering the negation of the property in question, and interpreting the result accordingly. To check whether there exists a path from s satisfying the LTL formula ϕ, we check whether all paths satisfy $\neg\phi$; a positive answer to this is a negative answer to our original question, and vice versa. We used this approach when analysing the ferryman puzzle in the previous section. However, as already noted, properties which *mix* universal and existential path quantifiers cannot in general be model checked using this approach, because the complement formula still has a mix.

Branching-time logics solve this problem by allowing us to quantify explicitly over paths. We will examine a logic known as *Computation Tree Logic*, or CTL. In CTL, as well as the temporal operators U, F, G and X of LTL we also have quantifiers A and E which express 'all paths' and 'exists a path', respectively. For example, we can write:

- There is a reachable state satisfying q: this is written $\mathrm{EF}\, q$.
- From all reachable states satisfying p, it is possible to maintain p continuously until reaching a state satisfying q: this is written $\mathrm{AG}\,(p \to \mathrm{E}[p \mathrel{\mathrm{U}} q])$.
- Whenever a state satisfying p is reached, the system can exhibit q continuously forevermore: $\mathrm{AG}\,(p \to \mathrm{EG}\, q)$.
- There is a reachable state from which all reachable states satisfy p: $\mathrm{EF}\,\mathrm{AG}\, p$.

3.4.1 Syntax of CTL

Computation Tree Logic, or CTL for short, is a *branching-time* logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the 'actual' path that is realised.

As before, we work with a fixed set of atomic formulas/descriptions (such as $p, q, r, \dots$, or $p_1, p_2, \dots$).

Definition 3.12 We define CTL formulas inductively via a Backus Naur form as done for LTL:

$$\begin{aligned}\phi ::= {} & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi \mid \\ & \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{A}[\phi \mathrel{\mathrm{U}} \phi] \mid \mathrm{E}[\phi \mathrel{\mathrm{U}} \phi]\end{aligned}$$

where p ranges over a set of atomic formulas.

Notice that each of the CTL temporal connectives is a pair of symbols. The first of the pair is one of A and E. A means 'along All paths' (*inevitably*) and E means 'along at least (there Exists) one path' (*possibly*). The second one of the pair is X, F, G, or U, meaning 'neXt state,' 'some Future state,' 'all future states (Globally)' and Until, respectively. The pair of symbols in $\mathrm{E}[\phi_1 \mathrel{\mathrm{U}} \phi_2]$, for example, is EU. In CTL, pairs of symbols like EU are indivisible. Notice that AU and EU are binary. The symbols X, F, G and U cannot occur without being preceded by an A or an E; similarly, every A or E must have one of X, F, G and U to accompany it.

Usually weak-until (W) and release (R) are not included in CTL, but they are derivable (see Section 3.4.5).

Convention 3.13 We assume similar binding priorities for the CTL connectives to what we did for propositional and predicate logic. The unary connectives (consisting of $\neg$ and the temporal connectives AG, EG, AF, EF, AX and EX) bind most tightly. Next in the order come $\wedge$ and $\vee$; and after that come $\rightarrow$, AU and EU.

Naturally, we can use brackets in order to override these priorities. Let us see some examples of well-formed CTL formulas and some examples which are not well-formed, in order to understand the syntax. Suppose that p, q and r are atomic formulas. The following are well-formed CTL formulas:

- $\mathrm{AG}\,(q \rightarrow \mathrm{EG}\,r)$, note that this is not the same as $\mathrm{AG}\,q \rightarrow \mathrm{EG}\,r$, for according to Convention 3.13, the latter formula means $(\mathrm{AG}\,q) \rightarrow (\mathrm{EG}\,r)$
- $\mathrm{EF}\,\mathrm{E}[r \mathrm{\ U\ } q]$
- $\mathrm{A}[p \mathrm{\ U\ } \mathrm{EF}\,r]$
- $\mathrm{EF}\,\mathrm{EG}\,p \rightarrow \mathrm{AF}\,r$, again, note that this binds as $(\mathrm{EF}\,\mathrm{EG}\,p) \rightarrow \mathrm{AF}\,r$, not $\mathrm{EF}\,(\mathrm{EG}\,p \rightarrow \mathrm{AF}\,r)$ or $\mathrm{EF}\,\mathrm{EG}\,(p \rightarrow \mathrm{AF}\,r)$
- $\mathrm{A}[p_1 \mathrm{\ U\ } \mathrm{A}[p_2 \mathrm{\ U\ } p_3]]$
- $\mathrm{E}[\mathrm{A}[p_1 \mathrm{\ U\ } p_2] \mathrm{\ U\ } p_3]$
- $\mathrm{AG}\,(p \rightarrow \mathrm{A}[p \mathrm{\ U\ } (\neg p \wedge \mathrm{A}[\neg p \mathrm{\ U\ } q])])$.

It is worth spending some time seeing how the syntax rules allow us to construct each of these. The following are not well-formed formulas:

- $\mathrm{EF}\,\mathrm{G}\,r$
- $\mathrm{A}\neg\mathrm{G}\,\neg p$
- $\mathrm{F}\,[r \mathrm{\ U\ } q]$
- $\mathrm{EF}\,(r \mathrm{\ U\ } q)$
- $\mathrm{AEF}\,r$
- $\mathrm{A}[(r \mathrm{\ U\ } q) \wedge (p \mathrm{\ U\ } r)]$.

It is especially worth understanding why the syntax rules don't allow us to construct these. For example, take $\mathrm{EF}\,(r \mathrm{\ U\ } q)$. The problem with this string is that U can occur only when paired with an A or an E. The E we have is paired with the F. To make this into a well-formed CTL formula, we would have to write $\mathrm{EF}\,\mathrm{E}[r \mathrm{\ U\ } q]$ or $\mathrm{EF}\,\mathrm{A}[r \mathrm{\ U\ } q]$.

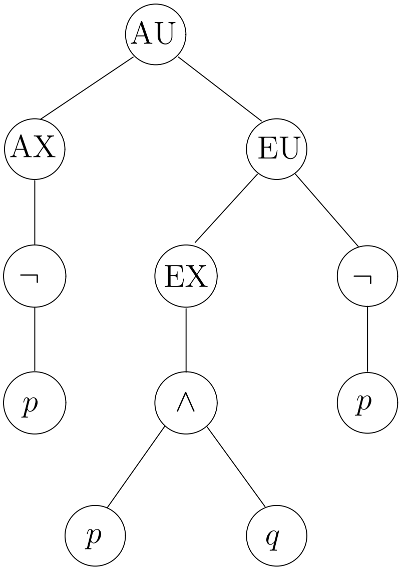


Figure 3.18. The parse tree of a CTL formula without infix notation.

Notice that we use square brackets after the A or E, when the paired operator is a U. There is no strong reason for this; you could use ordinary round brackets instead. However, it often helps one to read the formula (because we can more easily spot where the corresponding close bracket is). Another reason for using the square brackets is that SMV insists on it.

The reason $\mathrm{A}[(r \mathrel{\mathrm{U}} q) \wedge (p \mathrel{\mathrm{U}} r)]$ is not a well-formed formula is that the syntax does not allow us to put a boolean connective (like $\wedge$) directly inside A[ ] or E[ ]. Occurrences of A or E must be followed by one of G, F, X or U; when they are followed by U, it must be in the form $\mathrm{A}[\phi \mathrel{\mathrm{U}} \psi]$. Now, the ϕ and the ψ *may* contain $\wedge$, since they are arbitrary formulas; so $\mathrm{A}[(p \wedge q) \mathrel{\mathrm{U}} (\neg r \rightarrow q)]$ *is* a well-formed formula.

Observe that AU and EU are binary connectives which mix infix and prefix notation. In pure infix, we would write $\phi_1 \mathrel{\mathrm{AU}} \phi_2$, whereas in pure prefix we would write $\mathrm{AU}(\phi_1, \phi_2)$.

As with any formal language, and as we did in the previous two chapters, it is useful to draw parse trees for well-formed formulas. The parse tree for $\mathrm{A}[\mathrm{AX}\, \neg p \mathrel{\mathrm{U}} \mathrm{E}[\mathrm{EX}\,(p \wedge q) \mathrel{\mathrm{U}} \neg p]]$ is shown in Figure 3.18.

Definition 3.14 A subformula of a CTL formula ϕ is any formula ψ whose parse tree is a subtree of ϕ's parse tree.

3.4.2 Semantics of computation tree logic

CTL formulas are interpreted over transition systems (Definition 3.4). Let $\mathcal{M} = (S, \rightarrow, L)$ be such a model, $s \in S$ and ϕ a CTL formula. The definition of whether $\mathcal{M}, s \vDash \phi$ holds is recursive on the structure of ϕ, and can be roughly understood as follows:

- If ϕ is atomic, satisfaction is determined by L.
- If the top-level connective of ϕ (i.e., the connective occurring top-most in the parse tree of ϕ) is a boolean connective ($\wedge$, $\vee$, $\neg$, $\top$ etc.) then the satisfaction question is answered by the usual truth-table definition and further recursion down ϕ.
- If the top level connective is an operator beginning A, then satisfaction holds if all paths from s satisfy the 'LTL formula' resulting from removing the A symbol.
- Similarly, if the top level connective begins with E, then satisfaction holds if some path from s satisfy the 'LTL formula' resulting from removing the E.

In the last two cases, the result of removing A or E is not strictly an LTL formula, for it may contain further As or Es below. However, these will be dealt with by the recursion.

The formal definition of $\mathcal{M}, s \vDash \phi$ is a bit more verbose:

Definition 3.15 Let $\mathcal{M} = (S, \rightarrow, L)$ be a model for CTL, s in S, ϕ a CTL formula. The relation $\mathcal{M}, s \vDash \phi$ is defined by structural induction on ϕ:

1. $\mathcal{M}, s \vDash \top$ and $\mathcal{M}, s \nvDash \bot$
2. $\mathcal{M}, s \vDash p$ iff $p \in L(s)$
3. $\mathcal{M}, s \vDash \neg\phi$ iff $\mathcal{M}, s \nvDash \phi$
4. $\mathcal{M}, s \vDash \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ and $\mathcal{M}, s \vDash \phi_2$
5. $\mathcal{M}, s \vDash \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \vDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$
6. $\mathcal{M}, s \vDash \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, s \nvDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
7. $\mathcal{M}, s \vDash \mathrm{AX}\,\phi$ iff for all s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \vDash \phi$. Thus, AX says: 'in every next state.'
8. $\mathcal{M}, s \vDash \mathrm{EX}\,\phi$ iff for some s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \vDash \phi$. Thus, EX says: 'in some next state.' E is dual to A – in exactly the same way that $\exists$ is dual to $\forall$ in predicate logic.
9. $\mathcal{M}, s \vDash \mathrm{AG}\,\phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s, and all s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$. Mnemonically: for All computation paths beginning in s the property ϕ holds Globally. Note that 'along the path' includes the path's initial state s.
10. $\mathcal{M}, s \vDash \mathrm{EG}\,\phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s, and for all s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$. Mnemonically: there Exists a path beginning in s such that ϕ holds Globally along the path.

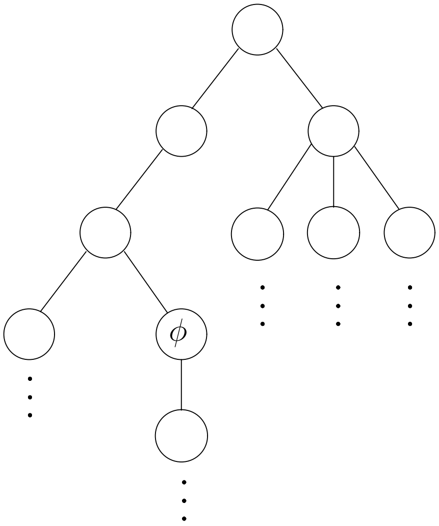


Figure 3.19. A system whose starting state satisfies EF ϕ.

11. $\mathcal{M}, s \vDash \mathrm{AF}\,\phi$ holds iff for all paths $s_1 \to s_2 \to \ldots$, where s_1 equals s, there is some s_i such that $\mathcal{M}, s_i \vDash \phi$. Mnemonically: for All computation paths beginning in s there will be some Future state where ϕ holds.
12. $\mathcal{M}, s \vDash \mathrm{EF}\,\phi$ holds iff there is a path $s_1 \to s_2 \to s_3 \to \ldots$, where s_1 equals s, and for some s_i along the path, we have $\mathcal{M}, s_i \vDash \phi$. Mnemonically: there Exists a computation path beginning in s such that ϕ holds in some Future state;
13. $\mathcal{M}, s \vDash \mathrm{A}[\phi_1 \mathrm{~U~} \phi_2]$ holds iff for all paths $s_1 \to s_2 \to s_3 \to \ldots$, where s_1 equals s, that path satisfies $\phi_1 \mathrm{~U~} \phi_2$, i.e., there is some s_i along the path, such that $\mathcal{M}, s_i \vDash \phi_2$, and, for each $j < i$, we have $\mathcal{M}, s_j \vDash \phi_1$. Mnemonically: All computation paths beginning in s satisfy that ϕ_1 Until ϕ_2 holds on it.
14. $\mathcal{M}, s \vDash \mathrm{E}[\phi_1 \mathrm{~U~} \phi_2]$ holds iff there is a path $s_1 \to s_2 \to s_3 \to \ldots$, where s_1 equals s, and that path satisfies $\phi_1 \mathrm{~U~} \phi_2$ as specified in 13. Mnemonically: there Exists a computation path beginning in s such that ϕ_1 Until ϕ_2 holds on it.

Clauses 9–14 above refer to computation paths in models. It is therefore useful to visualise all possible computation paths from a given state s by unwinding the transition system to obtain an infinite computation tree, whence ‘computation tree logic.’ The diagrams in Figures 3.19–3.22 show schematically systems whose starting states satisfy the formulas EF ϕ, EG ϕ, AG ϕ and AF ϕ, respectively. Of course, we could add more ϕ to any of these diagrams and still preserve the satisfaction – although there is nothing to add for AG . The diagrams illustrate a ‘least’ way of satisfying the formulas.

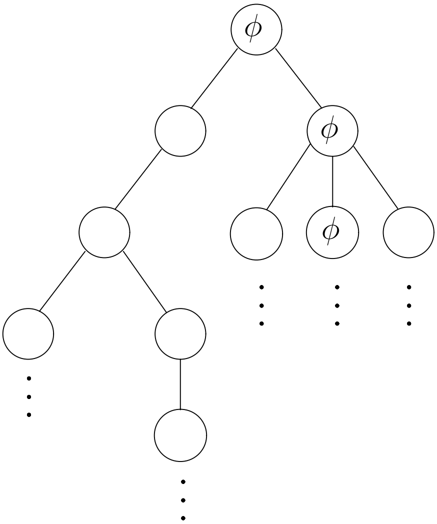


Figure 3.20. A system whose starting state satisfies EG ϕ.

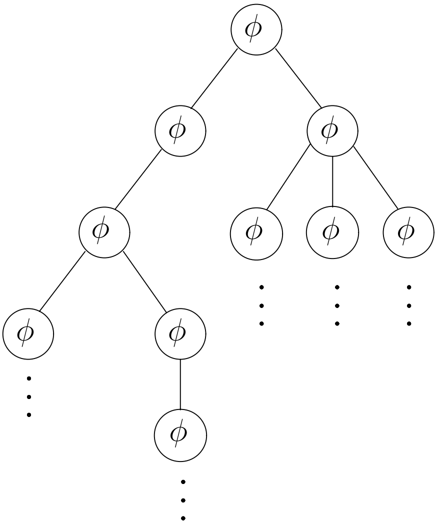


Figure 3.21. A system whose starting state satisfies AG ϕ.

Recall the transition system of Figure 3.3 for the designated starting state s_0, and the infinite tree illustrated in Figure 3.5. Let us now look at some example checks for this system.

1. $\mathcal{M}, s_0 \vDash p \wedge q$ holds since the atomic symbols p and q are contained in the node of s_0.
2. $\mathcal{M}, s_0 \vDash \neg r$ holds since the atomic symbol r is *not* contained in node s_0.

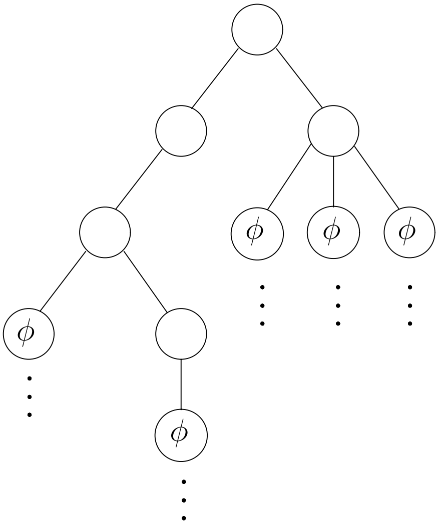


Figure 3.22. A system whose starting state satisfies AF ϕ.

3. $\mathcal{M}, s_0 \vDash \top$ holds by definition.
4. $\mathcal{M}, s_0 \vDash \mathrm{EX}\ (q \wedge r)$ holds since we have the leftmost computation path $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow \dots$ in Figure 3.5, whose second node s_1 contains q and r.
5. $\mathcal{M}, s_0 \vDash \neg\mathrm{AX}\ (q \wedge r)$ holds since we have the rightmost computation path $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ in Figure 3.5, whose second node s_2 only contains r, but *not* q.
6. $\mathcal{M}, s_0 \vDash \neg\mathrm{EF}\,(p \wedge r)$ holds since there is no computation path beginning in s_0 such that we could reach a state where $p \wedge r$ would hold. This is so because there is simply no state whatsoever in this system where p and r hold at the same time.
7. $\mathcal{M}, s_2 \vDash \mathrm{EG}\,r$ holds since there is a computation path $s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ beginning in s_2 such that r holds in all future states of that path; this is the only computation path beginning at s_2 and so $\mathcal{M}, s_2 \vDash \mathrm{AG}\,r$ holds as well.
8. $\mathcal{M}, s_0 \vDash \mathrm{AF}\,r$ holds since, for all computation paths beginning in s_0, the system reaches a state (s_1 or s_2) such that r holds.
9. $\mathcal{M}, s_0 \vDash \mathrm{E}[(p \wedge q)\ \mathrm{U}\ r]$ holds since we have the rightmost computation path $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ in Figure 3.5, whose second node s_2 $(i = 1)$ satisfies r, but all previous nodes (only $j = 0$, i.e., node s_0) satisfy $p \wedge q$.
10. $\mathcal{M}, s_0 \vDash \mathrm{A}[p\ \mathrm{U}\ r]$ holds since p holds at s_0 and r holds in any possible successor state of s_0, so $p\,\mathrm{U}\ r$ is true for all computation paths beginning in s_0 (so we may choose $i = 1$ independently of the path).
11. $\mathcal{M}, s_0 \vDash \mathrm{AG}\ (p \vee q \vee r \rightarrow \mathrm{EF}\ \mathrm{EG}\ r)$ holds since in all states reachable from s_0 and satisfying $p \vee q \vee r$ (all states in this case) the system can reach a state satisfying EG r (in this case state s_2).

3.4.3 Practical patterns of specifications

It's useful to look at some typical examples of formulas, and compare the situation with LTL (Section 3.2.3). Suppose atomic descriptions include some words such as busy and requested.

- It is possible to get to a state where started holds, but ready doesn't: $\mathrm{EF}\,(\mathsf{started} \wedge \neg\mathsf{ready})$. To express impossibility, we simply negate the formula.
- For any state, if a request (of some resource) occurs, then it will eventually be acknowledged:
  $\mathrm{AG}\,(\mathsf{requested} \to \mathrm{AF}\ \mathsf{acknowledged})$.
- The property that if the process is enabled infinitely often, then it runs infinitely often, is not expressible in CTL. In particular, it is not expressed by $\mathrm{AG}\,\mathrm{AF}\ \mathsf{enabled} \to \mathrm{AG}\,\mathrm{AF}\ \mathsf{running}$, or indeed any other insertion of A or E into the corresponding LTL formula. The CTL formula just given expresses that if every path has infinitely often enabled, then every path is infinitely often taken; this is much weaker than asserting that every path which has infinitely often enabled is infinitely often taken.
- A certain process is enabled infinitely often on every computation path:
  $\mathrm{AG}\,(\mathrm{AF}\ \mathsf{enabled})$.
- Whatever happens, a certain process will eventually be permanently deadlocked:
  $\mathrm{AF}\,(\mathrm{AG}\ \mathsf{deadlock})$.
- From any state it is possible to get to a restart state:
  $\mathrm{AG}\,(\mathrm{EF}\ \mathsf{restart})$.
- An upwards travelling lift at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
  $\mathrm{AG}\,(\mathsf{floor2} \wedge \mathsf{directionup} \wedge \mathsf{ButtonPressed5} \to \mathrm{A}[\mathsf{directionup}\ \mathrm{U}\ \mathsf{floor5}])$
  Here, our atomic descriptions are boolean expressions built from system variables, e.g., floor2.
- The lift can remain idle on the third floor with its doors closed:
  $\mathrm{AG}\,(\mathsf{floor3} \wedge \mathsf{idle} \wedge \mathsf{doorclosed} \to \mathrm{EG}\,(\mathsf{floor3} \wedge \mathsf{idle} \wedge \mathsf{doorclosed}))$.
- A process can always request to enter its critical section. Recall that this was not expressible in LTL. Using the propositions of Figure 3.8, this may be written $\mathrm{AG}\,(n_1 \to \mathrm{EX}\,t_1)$ in CTL.
- Processes need not enter their critical section in strict sequence. This was also not expressible in LTL, though we expressed its negation. CTL allows us to express it directly: $\mathrm{EF}\,(c_1 \wedge \mathrm{E}[c_1\ \mathrm{U}\ (\neg c_1 \wedge \mathrm{E}[\neg c_2\ \mathrm{U}\ c_1])])$.

3.4.4 Important equivalences between CTL formulas

Definition 3.16 Two CTL formulas ϕ and ψ are said to be semantically equivalent if any state in any model which satisfies one of them also satisfies the other; we denote this by $\phi \equiv \psi$.

We have already noticed that A is a universal quantifier on paths and E is the corresponding existential quantifier. Moreover, G and F are also universal and existential quantifiers, ranging over the states along a particular path. In view of these facts, it is not surprising to find that de Morgan rules exist:

$$\begin{aligned} \neg \mathrm{AF}\,\phi &\equiv \mathrm{EG}\,\neg\phi \\ \neg \mathrm{EF}\,\phi &\equiv \mathrm{AG}\,\neg\phi \\ \neg \mathrm{AX}\,\phi &\equiv \mathrm{EX}\,\neg\phi. \end{aligned} \tag{3.6}$$

We also have the equivalences

$$\mathrm{AF}\,\phi \;\equiv\; \mathrm{A}[\top \;\mathrm{U}\; \phi] \qquad \mathrm{EF}\,\phi \;\equiv\; \mathrm{E}[\top \;\mathrm{U}\; \phi]$$

which are similar to the corresponding equivalences in LTL.

3.4.5 Adequate sets of CTL connectives

As in propositional logic and in LTL, there is some redundancy among the CTL connectives. For example, the connective AX can be written $\neg$EX $\neg$; and AG, AF, EG and EF can be written in terms of AU and EU as follows: first, write AG ϕ as $\neg$EF $\neg\phi$ and EG ϕ as $\neg$AF $\neg\phi$, using (3.6), and then use AF $\phi \equiv \mathrm{A}[\top \;\mathrm{U}\; \phi]$ and EF $\phi \equiv \mathrm{E}[\top \;\mathrm{U}\; \phi]$. Therefore AU, EU and EX form an adequate set of temporal connectives.

Also EG, EU, and EX form an adequate set, for we have the equivalence

$$\mathrm{A}[\phi \;\mathrm{U}\; \psi] \;\equiv\; \neg(\mathrm{E}[\neg\psi \;\mathrm{U}\; (\neg\phi \wedge \neg\psi)] \vee \mathrm{EG}\,\neg\psi) \tag{3.7}$$

which can be proved as follows:

$$\begin{aligned} \mathrm{A}[\phi \;\mathrm{U}\; \psi] &\equiv \mathrm{A}[\neg(\neg\psi \;\mathrm{U}\; (\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi] \\ &\equiv \neg\mathrm{E}\neg[\neg(\neg\psi \;\mathrm{U}\; (\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi] \\ &\equiv \neg\mathrm{E}[(\neg\psi \;\mathrm{U}\; (\neg\phi \wedge \neg\psi)) \vee \mathrm{G}\,\neg\psi] \\ &\equiv \neg(\mathrm{E}[\neg\psi \;\mathrm{U}\; (\neg\phi \wedge \neg\psi)] \vee \mathrm{EG}\,\neg\psi). \end{aligned}$$

The first line is by Theorem 3.10, and the remainder by elementary manipulation. (This proof involves intermediate formulas which violate the syntactic formation rules of CTL; however, it is valid in the logic CTL* introduced in the next section.) More generally, we have:

Theorem 3.17 A set of temporal connectives in CTL is adequate if, and only if, it contains at least one of {AX , EX }, at least one of {EG , AF , AU } and EU .

This theorem is proved in a paper referenced in the bibliographic notes at the end of the chapter. The connective EU plays a special role in that theorem because neither weak-until W nor release R are primitive in CTL (Definition 3.12). The temporal connectives AR, ER, AW and EW are all definable in CTL:

- $\mathrm{A}[\phi\ \mathrm{R}\ \psi] = \neg\mathrm{E}[\neg\phi\ \mathrm{U}\ \neg\psi]$
- $\mathrm{E}[\phi\ \mathrm{R}\ \psi] = \neg\mathrm{A}[\neg\phi\ \mathrm{U}\ \neg\psi]$
- $\mathrm{A}[\phi\ \mathrm{W}\ \psi] = \mathrm{A}[\psi\ \mathrm{R}\ (\phi \vee \psi)]$, and then use the first equation above
- $\mathrm{E}[\phi\ \mathrm{W}\ \psi] = \mathrm{E}[\psi\ \mathrm{R}\ (\phi \vee \psi)]$, and then use the second one.

These definitions are justified by LTL equivalences in Sections 3.2.4 and 3.2.5. Some other noteworthy equivalences in CTL are the following:

$$\begin{aligned}
\mathrm{AG}\,\phi &\equiv \phi \wedge \mathrm{AX}\,\mathrm{AG}\,\phi \\
\mathrm{EG}\,\phi &\equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi \\
\mathrm{AF}\,\phi &\equiv \phi \vee \mathrm{AX}\,\mathrm{AF}\,\phi \\
\mathrm{EF}\,\phi &\equiv \phi \vee \mathrm{EX}\,\mathrm{EF}\,\phi \\
\mathrm{A}[\phi\ \mathrm{U}\ \psi] &\equiv \psi \vee (\phi \wedge \mathrm{AX}\,\mathrm{A}[\phi\ \mathrm{U}\ \psi]) \\
\mathrm{E}[\phi\ \mathrm{U}\ \psi] &\equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi\ \mathrm{U}\ \psi]).
\end{aligned}$$

For example, the intuition for the third one is the following: in order to have $\mathrm{AF}\,\phi$ in a particular state, ϕ must be true at some point along each path from that state. To achieve this, we either have ϕ true now, in the current state; or we postpone it, in which case we must have $\mathrm{AF}\,\phi$ in each of the next states. Notice how this equivalence appears to define AF in terms of AX and AF itself, an apparently circular definition. In fact, these equivalences can be used to define the six connectives on the left in terms of AX and EX, in a *non-circular* way. This is called the fixed-point characterisation of CTL; it is the mathematical foundation for the model-checking algorithm developed in Section 3.6.1; and we return to it later (Section 3.7).

3.5 CTL* and the expressive powers of LTL and CTL

CTL allows explicit quantification over paths, and in this respect it is more expressive than LTL, as we have seen. However, it does not allow one to select a range of paths by describing them with a formula, as LTL does. In that respect, LTL is more expressive. For example, in LTL we can say 'all paths which have a p along them also have a q along them,' by writing $\mathrm{F}\,p \to \mathrm{F}\,q$. It is not possible to write this in CTL because of the constraint that every F has an associated A or E. The formula $\mathrm{AF}\,p \to \mathrm{AF}\,q$ means something quite different: it says 'if all paths have a p along them, then all paths have a q along them.' One might write AG $(p \rightarrow \mathrm{AF}\ q)$, which is closer, since it says that every way of extending every path to a p eventually meets a q, but that is still not capturing the meaning of $\mathrm{F}\,p \rightarrow \mathrm{F}\,q$.

CTL* is a logic which combines the expressive powers of LTL and CTL, by dropping the CTL constraint that every temporal operator (X, U, F, G) has to be associated with a unique path quantifier (A, E). It allows us to write formulas such as

- $\mathrm{A}[(p \mathrel{\mathrm{U}} r) \vee (q \mathrel{\mathrm{U}} r)]$: along all paths, either p is true until r, or q is true until r.
- $\mathrm{A}[X\,p \vee \mathrm{X}\,\mathrm{X}\,p]$: along all paths, p is true in the next state, or the next but one.
- $\mathrm{E}[\mathrm{G}\,\mathrm{F}\,p]$: there is a path along which p is infinitely often true.

These formulas are *not* equivalent to, respectively, $\mathrm{A}[(p \vee q) \mathrel{\mathrm{U}} r)]$, $\mathrm{AX}\,p \vee \mathrm{AX}\,\mathrm{AX}\,p$ and $\mathrm{EG}\,\mathrm{EF}\,p$. It turns out that the first of them can be written as a (rather long) CTL formula. The second and third do not have a CTL equivalent.

The syntax of CTL* involves two classes of formulas:

- *state formulas*, which are evaluated in states:

$$\phi ::= \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha]$$

  where p is any atomic formula and α any path formula; and
- *path formulas*, which are evaluated along paths:

$$\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \mathrel{\mathrm{U}} \alpha) \mid (\mathrm{G}\,\alpha) \mid (\mathrm{F}\,\alpha) \mid (\mathrm{X}\,\alpha)$$

where ϕ is any state formula. This is an example of an inductive definition which is *mutually recursive*: the definition of each class depends upon the definition of the other, with base cases p and $\top$.

LTL and CTL as subsets of CTL* Although the syntax of LTL does not include A and E, the semantic viewpoint of LTL is that we consider all paths. Therefore, the LTL formula α is equivalent to the CTL* formula $\mathrm{A}[\alpha]$. Thus, LTL can be viewed as a subset of CTL*.

CTL is also a subset of CTL*, since it is the fragment of CTL* in which we restrict the form of path formulas to

$$\alpha ::= (\phi \mathrel{\mathrm{U}} \phi) \mid (\mathrm{G}\,\phi) \mid (\mathrm{F}\,\phi) \mid (\mathrm{X}\,\phi)$$

Figure 3.23 shows the relationship among the expressive powers of CTL, LTL and CTL*. Here are some examples of formulas in each of the subsets

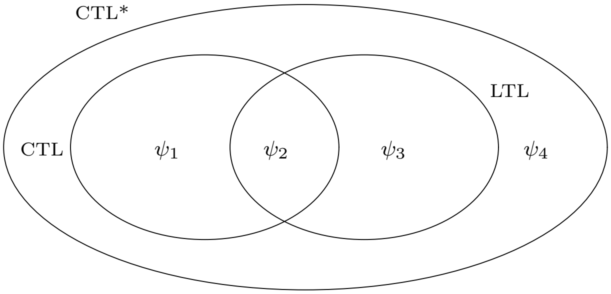
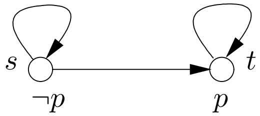


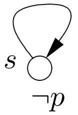
Figure 3.23. The expressive powers of CTL, LTL and CTL*.

shown:

In CTL but not in LTL: $\psi_1 \overset{\text{def}}{=} \mathrm{AG\,EF}\,p$. This expresses: wherever we have got to, we can always get to a state in which p is true. This is also useful, e.g., in finding deadlocks in protocols.

The proof that $\mathrm{AG\,EF}\,p$ is not expressible in LTL is as follows. Let ϕ be an LTL formula such that $\mathrm{A}[\phi]$ is allegedly equivalent to $\mathrm{AG\,EF}\,p$. Since $\mathcal{M}, s \vDash \mathrm{AG\,EF}\,p$ in the left-hand diagram below, we have $\mathcal{M}, s \vDash \mathrm{A}[\phi]$. Now let $\mathcal{M}'$ be as shown in the right-hand diagram. The paths from s in $\mathcal{M}'$ are a subset of those from s in $\mathcal{M}$, so we have $\mathcal{M}', s \vDash \mathrm{A}[\phi]$. Yet, it is *not* the case that $\mathcal{M}', s \vDash \mathrm{AG\,EF}\,p$; a contradiction.





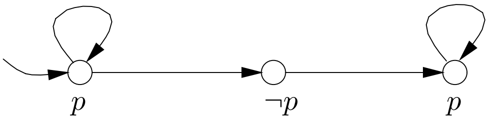
In CTL*, but neither in CTL nor in LTL: $\psi_4 \overset{\text{def}}{=} \mathrm{E}[\mathrm{G\,F}\,p]$, saying that there is a path with infinitely many p.
The proof that this is not expressible in CTL is quite complex and may be found in the papers co-authored by E. A. Emerson with others, given in the references. (Why is it not expressible in LTL?)

In LTL but not in CTL: $\psi_3 \overset{\text{def}}{=} \mathrm{A}[\mathrm{G\,F}\,p \to \mathrm{F}\,q]$, saying that if there are infinitely many p along the path, then there is an occurrence of q. This is an interesting thing to be able to say; for example, many fairness constraints are of the form 'infinitely often requested implies eventually acknowledged'.

In LTL and CTL: $\psi_2 \overset{\text{def}}{=} \mathrm{AG}\,(p \to \mathrm{AF}\,q)$ in CTL, or $\mathrm{G}\,(p \to \mathrm{F}\,q)$ in LTL: any p is eventually followed by a q.

Remark 3.18 We just saw that some (but not all) LTL formulas can be converted into CTL formulas by adding an A to each temporal operator. For a positive example, the LTL formula $\mathrm{G}\,(p \to \mathrm{F}\,q)$ is equivalent to the CTL formula $\mathrm{AG}\ (p \to \mathrm{AF}\ q)$. We discuss two more negative examples:

- $\mathrm{F\,G}\,p$ and $\mathrm{AF\ AG}\ p$ are not equivalent, since $\mathrm{F\,G}\,p$ is satisfied, whereas $\mathrm{AF\ AG}\ p$ is not satisfied, in the model

  In fact, $\mathrm{AF\ AG}\ p$ is strictly stronger than $\mathrm{F\,G}\,p$.
- While the LTL formulas $\mathrm{X\,F}\,p$ and $\mathrm{F\,X}\,p$ are equivalent, and they are equivalent to the CTL formula $\mathrm{AX\ AF}\ p$, they are not equivalent to $\mathrm{AF\ AX}\ p$. The latter is strictly stronger, and has quite a strange meaning (try working it out).

Remark 3.19 There is a considerable literature comparing linear-time and branching-time logics. The question of which one is 'better' has been debated for about 20 years. We have seen that they have incomparable expressive powers. CTL* is more expressive than either of them, but is computationally much more expensive (as will be seen in Section 3.6). The choice between LTL and CTL depends on the application at hand, and on personal preference. LTL lacks CTL's ability to quantify over paths, and CTL lacks LTL's finer-grained ability to describe individual paths. To many people, LTL appears to be more straightforward to use; as noted above, CTL formulas like $\mathrm{AF\,AX}\,p$ seem hard to understand.

3.5.1 Boolean combinations of temporal formulas in CTL

Compared with CTL*, the syntax of CTL is restricted in two ways: it does not allow boolean combinations of path formulas and it does not allow nesting of the path modalities X, F and G. Indeed, we have already seen examples of the inexpressibility in CTL of nesting of path modalities, namely the formulas ψ_3 and ψ_4 above.

In this section, we see that the first of these restrictions is only apparent; we can find equivalents in CTL for formulas having boolean combinations of path formulas. The idea is to translate any CTL formula having boolean combinations of path formulas into a CTL formula that doesn't. For example, we may see that $\mathrm{E}[\mathrm{F}\,p \wedge \mathrm{F}\,q] \ \equiv\ \mathrm{EF}\,[p \wedge \mathrm{EF}\,q] \vee \mathrm{EF}\,[q \wedge \mathrm{EF}\,p]$ since, if we have $\mathrm{F}\,p \wedge \mathrm{F}\,q$ along any path, then either the p must come before the q, or the other way around, corresponding to the two disjuncts on the right. (If the p and q occur simultaneously, then both disjuncts are true.)

Since U is like F (only with the extra complication of its first argument), we find the following equivalence:

$$\begin{aligned}\mathrm{E}[(p_1 \mathrm{~U~} q_1) \wedge (p_2 \mathrm{~U~} q_2)] \equiv\ & \mathrm{E}[(p_1 \wedge p_2) \mathrm{~U~} (q_1 \wedge \mathrm{E}[p_2 \mathrm{~U~} q_2])] \\ & \vee \mathrm{E}[(p_1 \wedge p_2) \mathrm{~U~} (q_2 \wedge \mathrm{E}[p_1 \mathrm{~U~} q_1])].\end{aligned}$$

And from the CTL equivalence $\mathrm{A}[p \mathrm{~U~} q] \equiv \neg(\mathrm{E}[\neg q \mathrm{~U~} (\neg p \wedge \neg q)] \vee \mathrm{EG}\, \neg q)$ (see Theorem 3.10) we can obtain $\mathrm{E}[\neg(p \mathrm{~U~} q)] \equiv \mathrm{E}[\neg q \mathrm{~U~} (\neg p \wedge \neg q)] \vee \mathrm{EG}\, \neg q$. Other identities we need in this translation include $\mathrm{E}[\neg \mathrm{X}\, p] \equiv \mathrm{EX}\, \neg p$.

3.5.2 Past operators in LTL

The temporal operators X, U, F, etc. which we have seen so far refer to the future. Sometimes we want to encode properties that refer to the past, such as: ‘whenever q occurs, then there was some p in the past.’ To do this, we may add the operators Y, S, O, H. They stand for *yesterday*, *since*, *once*, and *historically*, and are the past analogues of X, U, F, G, respectively. Thus, the example formula may be written $\mathrm{G}\, (q \to O\, p)$.

NuSMV supports past operators in LTL. One could also add past operators to CTL (AY, ES, etc.) but NuSMV does not support them.

Somewhat counter-intuitively, past operators do not increase the expressive power of LTL. That is to say, every LTL formula with past operators can be written equivalently without them. The example formula above can be written $\neg p \mathrm{~W~} q$, or equivalently $\neg(\neg q \mathrm{~U~} (p \wedge \neg q))$ if one wants to avoid W. This result is surprising, because it seems that being able to talk about the past as well as the future allows more expressivity than talking about the future alone. However, recall that LTL equivalence is quite crude: it says that the two formulas are satisfied by exactly the same set of paths. The past operators allow us to travel backwards along the path, but only to reach points we could have reached by travelling forwards from its beginning. In contrast, adding past operators to CTL does increase its expressive power, because they can allow us to examine states not forward-reachable from the present one.

3.6 Model-checking algorithms

The semantic definitions for LTL and CTL presented in Sections 3.2 and 3.4 allow us to test whether the initial states of a given system satisfy an LTL or CTL formula. This is the basic model-checking question. In general, interesting transition systems will have a huge number of states and the formula we are interested in checking may be quite long. It is therefore well worth trying to find efficient algorithms.

Although LTL is generally preferred by specifiers, as already noted, we start with CTL model checking because its algorithm is simpler.

3.6.1 The CTL model-checking algorithm

Humans may find it easier to do model checks on the unwindings of models into infinite trees, given a designated initial state, for then all possible paths are plainly visible. However, if we think of implementing a model checker on a computer, we certainly cannot unwind transition systems into infinite trees. We need to do checks on *finite* data structures. For this reason, we now have to develop new insights into the semantics of CTL. Such a deeper understanding will provide the basis for an efficient algorithm which, given $\mathcal{M}$, $s \in S$ and ϕ, computes whether $\mathcal{M}, s \vDash \phi$ holds. In the case that ϕ is not satisfied, such an algorithm can be augmented to produce an actual path (= run) of the system demonstrating that $\mathcal{M}$ cannot satisfy ϕ. That way, we may *debug* a system by trying to fix what enables runs which refute ϕ.

There are various ways in which one could consider

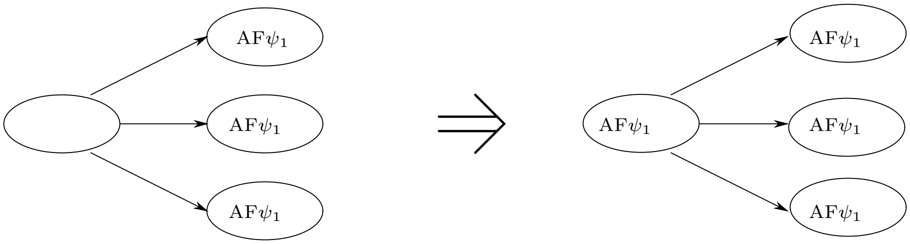
$$\mathcal{M}, s_0 \overset{?}{\vDash} \phi$$

as a computational problem. For example, one could have the model $\mathcal{M}$, the formula ϕ and a state s_0 as input; one would then expect a reply of the form 'yes' ($\mathcal{M}, s_0 \vDash \phi$ holds), or 'no' ($\mathcal{M}, s_0 \vDash \phi$ does not hold). Alternatively, the inputs could be just $\mathcal{M}$ and ϕ, where the output would be *all* states s of the model $\mathcal{M}$ which satisfy ϕ.

It turns out that it is easier to provide an algorithm for solving the second of these two problems. This automatically gives us a solution to the first one, since we can simply check whether s_0 is an element of the output set.

The labelling algorithm We present an algorithm which, given a model and a CTL formula, outputs the set of states of the model that satisfy the formula. The algorithm does not need to be able to handle every CTL connective explicitly, since we have already seen that the connectives $\bot$, $\neg$ and $\wedge$ form an adequate set as far as the propositional connectives are concerned; and AF , EU and EX form an adequate set of temporal connectives. Given an arbitrary CTL formula ϕ, we would simply pre-process ϕ in order to write it in an equivalent form in terms of the adequate set of connectives, and then

Repeat…



… until no change.

Figure 3.24. The iteration step of the procedure for labelling states with subformulas of the form AF ψ_1.

call the model-checking algorithm. Here is the algorithm:

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula ϕ.
OUTPUT: the set of states of $\mathcal{M}$ which satisfy ϕ.

First, change ϕ to the output of TRANSLATE (ϕ), i.e., we write ϕ in terms of the connectives AF, EU, EX, $\wedge$, $\neg$ and $\bot$ using the equivalences given earlier in the chapter. Next, label the states of $\mathcal{M}$ with the subformulas of ϕ that are satisfied there, starting with the smallest subformulas and working outwards towards ϕ.

Suppose ψ is a subformula of ϕ and states satisfying all the *immediate* subformulas of ψ have already been labelled. We determine by a case analysis which states to label with ψ. If ψ is

- $\bot$: then no states are labelled with $\bot$.
- p: then label s with p if $p \in L(s)$.
- $\psi_1 \wedge \psi_2$: label s with $\psi_1 \wedge \psi_2$ if s is already labelled both with ψ_1 and with ψ_2.
- $\neg\psi_1$: label s with $\neg\psi_1$ if s is not already labelled with ψ_1.
- AF ψ_1:
  - If any state s is labelled with ψ_1, label it with AF ψ_1.
  - Repeat: label any state with AF ψ_1 if all successor states are labelled with AF ψ_1, until there is no change. This step is illustrated in Figure 3.24.
- E[ψ_1 U ψ_2]:
  - If any state s is labelled with ψ_2, label it with E[ψ_1 U ψ_2].
  - Repeat: label any state with E[ψ_1 U ψ_2] if it is labelled with ψ_1 and at least one of its successors is labelled with E[ψ_1 U ψ_2], until there is no change. This step is illustrated in Figure 3.25.
- EX ψ_1: label any state with EX ψ_1 if one of its successors is labelled with ψ_1.

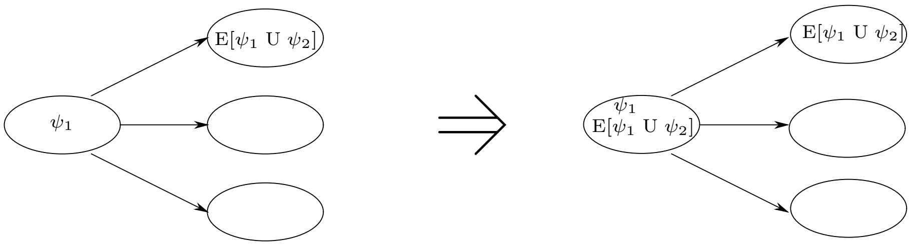

$E[\psi_1 \text{ U } \psi_2]$
ψ_1
ψ_1
$E[\psi_1 \text{ U } \psi_2]$
$E[\psi_1 \text{ U } \psi_2]$

Figure 3.25. The iteration step of the procedure for labelling states with subformulas of the form $E[\psi_1 \text{ U } \psi_2]$.

Having performed the labelling for all the subformulas of ϕ (including ϕ itself), we output the states which are labelled ϕ.

The complexity of this algorithm is $O(f \cdot V \cdot (V + E))$, where f is the number of connectives in the formula, V is the number of states and E is the number of transitions; the algorithm is linear in the size of the formula and quadratic in the size of the model.

Handling EG **directly** Instead of using a minimal adequate set of connectives, it would have been possible to write similar routines for the other connectives. Indeed, this would probably be more efficient. The connectives AG and EG require a slightly different approach from that for the others, however. Here is the algorithm to deal with $\text{EG}\,\psi_1$ *directly*:

- $\text{EG}\,\psi_1$:
  - Label *all* the states with $\text{EG}\,\psi_1$.
  - If any state s is *not* labelled with ψ_1, *delete* the label $\text{EG}\,\psi_1$.
  - Repeat: *delete* the label $\text{EG}\,\psi_1$ from any state if *none* of its successors is labelled with $\text{EG}\,\psi_1$; until there is no change.

Here, we label all the states with the subformula $\text{EG}\,\psi_1$ and then whittle down this labelled set, instead of building it up from nothing as we did in the case for EU. Actually, there is no real difference between this procedure for $\text{EG}\,\psi$ and what you would do if you translated it into $\neg\text{AF}\,\neg\psi$ as far as the final result is concerned.

A variant which is more efficient We can improve the efficiency of our labelling algorithm by using a cleverer way of handling EG. Instead of using EX, EU and AF as the adequate set, we use EX, EU and EG instead. For EX and EU we do as before (but take care to search the model by

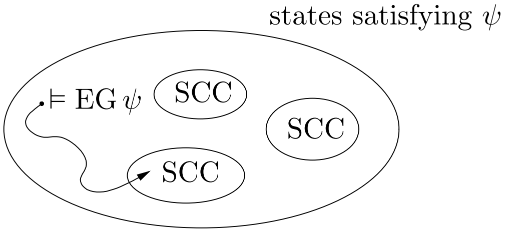


Figure 3.26. A better way of handling EG.

backwards breadth-first search, for this ensures that we won't have to pass over any node twice). For the EG ψ case:

- Restrict the graph to states satisfying ψ, i.e., delete all other states and their transitions;
- Find the maximal *strongly connected components* (SCCs); these are maximal regions of the state space in which every state is linked with (= has a finite path to) every other one in that region.
- Use backwards breadth-first search on the restricted graph to find any state that can reach an SCC; see Figure 3.26.

The complexity of this algorithm is $O(f \cdot (V + E))$, i.e., linear both in the size of the model and in the size of the formula.

Example 3.20 We applied the basic algorithm to our second model of mutual exclusion with the formula $\mathrm{E}[\neg c_2 \mathrm{~U~} c_1]$; see Figure 3.27. The algorithm labels all states which satisfy c_1 during phase 1 with $\mathrm{E}[\neg c_2 \mathrm{~U~} c_1]$. This labels s_2 and s_4. During phase 2, it labels all states which do not satisfy c_2 and have a successor state that is already labelled. This labels states s_1 and s_3. During phase 3, we label s_0 because it does not satisfy c_2 and has a successor state (s_1) which is already labelled. Thereafter, the algorithm terminates because no additional states get labelled: all unlabelled states either satisfy c_2, or must pass through such a state to reach a labelled state.

The pseudo-code of the CTL model-checking algorithm We present the pseudo-code for the basic labelling algorithm. The main function `SAT` (for 'satisfies') takes as input a CTL formula. The program `SAT` expects a parse tree of some CTL formula constructed by means of the grammar in Definition 3.12. This expectation reflects an important *precondition* on the correctness of the algorithm `SAT`. For example, the program simply would not know what to do with an input of the form $\mathrm{X}\,(\top \wedge \mathrm{EF}\, p_3)$, since this is not a CTL formula.

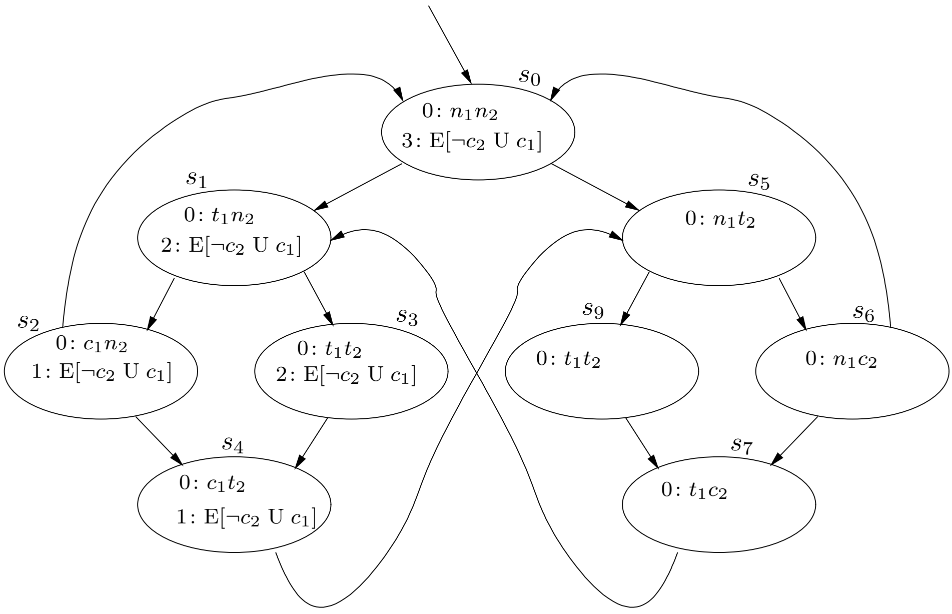


Figure 3.27. An example run of the labelling algorithm in our second model of mutual exclusion applied to the formula E[$\neg c_2$ U c_1].

The pseudo-code we write for `SAT` looks a bit like fragments of C or Java code; we use functions with a keyword **return** that indicates which result the function should return. We will also use natural language to indicate the case analysis over the root node of the parse tree of ϕ. The declaration **local var** declares some fresh variables local to the current instance of the procedure in question, whereas **repeat until** executes the command which follows it repeatedly, until the condition becomes true. Additionally, we employ suggestive notation for the operations on sets, like intersection, set complement and so forth. In reality we would need an abstract data type, together with implementations of these operations, but for now we are interested only in the mechanism in principle of the algorithm for `SAT`; any (correct and efficient) implementation of sets would do and we study such an implementation in Chapter 6. We assume that `SAT` has access to all the relevant parts of the model: S, $\rightarrow$ and L. In particular, we ignore the fact that `SAT` would require a description of $\mathcal{M}$ as input as well. We simply assume that `SAT` operates *directly* on any such given model. Note how `SAT` translates ϕ into an equivalent formula of the adequate set chosen.

```
function SAT ($\phi$)
 /* determines the set of states satisfying $\phi$ */
begin
    case
        $\phi$ is $\top$ : return $S$
        $\phi$ is $\bot$ : return $\emptyset$
        $\phi$ is atomic: return $\{s \in S \mid \phi \in L(s)\}$
        $\phi$ is $\neg\phi_1$ : return $S -$ SAT ($\phi_1$)
        $\phi$ is $\phi_1 \wedge \phi_2$ : return SAT ($\phi_1$) $\cap$ SAT ($\phi_2$)
        $\phi$ is $\phi_1 \vee \phi_2$ : return SAT ($\phi_1$) $\cup$ SAT ($\phi_2$)
        $\phi$ is $\phi_1 \rightarrow \phi_2$ : return SAT ($\neg\phi_1 \vee \phi_2$)
        $\phi$ is $\mathrm{AX}\,\phi_1$ : return SAT ($\neg\mathrm{EX}\,\neg\phi_1$)
        $\phi$ is $\mathrm{EX}\,\phi_1$ : return SAT$_{\mathtt{EX}}$($\phi_1$)
        $\phi$ is $\mathrm{A}[\phi_1\ \mathrm{U}\ \phi_2]$ : return SAT($\neg(\mathrm{E}[\neg\phi_2\ \mathrm{U}\ (\neg\phi_1 \wedge \neg\phi_2)] \vee \mathrm{EG}\,\neg\phi_2)$)
        $\phi$ is $\mathrm{E}[\phi_1\ \mathrm{U}\ \phi_2]$ : return SAT$_{\mathtt{EU}}$($\phi_1, \phi_2$)
        $\phi$ is $\mathrm{EF}\,\phi_1$ : return SAT ($\mathrm{E}(\top\ \mathrm{U}\ \phi_1)$)
        $\phi$ is $\mathrm{EG}\,\phi_1$ : return SAT($\neg\mathrm{AF}\,\neg\phi_1$)
        $\phi$ is $\mathrm{AF}\,\phi_1$ : return SAT$_{\mathtt{AF}}$ ($\phi_1$)
        $\phi$ is $\mathrm{AG}\,\phi_1$ : return SAT ($\neg\mathrm{EF}\,\neg\phi_1$)
    end case
end function
```

Figure 3.28. The function SAT. It takes a CTL formula as input and returns the set of states satisfying the formula. It calls the functions SAT$_{\mathtt{EX}}$, SAT$_{\mathtt{EU}}$ and SAT$_{\mathtt{AF}}$, respectively, if EX, EU or AF is the root of the input's parse tree.

The algorithm is presented in Figure 3.28 and its subfunctions in Figures 3.29–3.31. They use program variables X, Y, V and W which are sets of states. The program for SAT handles the easy cases directly and passes more complicated cases on to special procedures, which in turn might call SAT *recursively* on subexpressions. These special procedures rely on implementations of the functions

$$\begin{aligned}\mathrm{pre}_\exists(Y) &= \{s \in S \mid \text{exists } s', (s \rightarrow s' \text{ and } s' \in Y)\}\\ \mathrm{pre}_\forall(Y) &= \{s \in S \mid \text{for all } s', (s \rightarrow s' \text{ implies } s' \in Y)\}.\end{aligned}$$

'Pre' denotes travelling backwards along the transition relation. Both functions compute a pre-image of a set of states. The function $\mathrm{pre}_\exists$ (instrumental in SAT$_{\mathtt{EX}}$ and SAT$_{\mathtt{EU}}$) takes a subset Y of states and returns the set of states which *can* make a transition into Y. The function $\mathrm{pre}_\forall$, used in SAT$_{\mathtt{AF}}$, takes

```
function SAT_EX ($\phi$)
 /* determines the set of states satisfying EX $\phi$ */
local var $X, Y$
begin
    $X$ := SAT ($\phi$);
    $Y$ := $\mathrm{pre}_\exists(X)$;
    return $Y$
end
```

Figure 3.29. The function $\mathtt{SAT}_{\mathtt{EX}}$. It computes the states satisfying ϕ by calling SAT. Then, it looks backwards along → to find the states satisfying $\mathrm{EX}\,\phi$.

```
function SAT_AF ($\phi$)
 /* determines the set of states satisfying AF $\phi$ */
local var $X, Y$
begin
    $X$ := $S$;
    $Y$ := SAT ($\phi$);
    repeat until $X = Y$
    begin
        $X$ := $Y$;
        $Y$ := $Y \cup \mathrm{pre}_\forall(Y)$
    end
    return $Y$
end
```

Figure 3.30. The function $\mathtt{SAT}_{\mathtt{AF}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying $\mathrm{AF}\,\phi$ in the manner described in the labelling algorithm.

a set Y and returns the set of states which make transitions *only* into Y. Observe that $\mathrm{pre}_\forall$ can be expressed in terms of complementation and $\mathrm{pre}_\exists$, as follows:

$$\mathrm{pre}_\forall(Y) = S - \mathrm{pre}_\exists(S - Y) \tag{3.8}$$

where we write $S - Y$ for the set of all $s \in S$ which are not in Y.

The correctness of this pseudocode and the model checking algorithm is discussed in Section 3.7.

```
function SAT_EU ($\phi, \psi$)
 /* determines the set of states satisfying $\mathrm{E}[\phi \mathrm{\ U\ } \psi]$ */
local var $W, X, Y$
begin
    $W :=$ SAT ($\phi$);
    $X := S$;
    $Y :=$ SAT ($\psi$);
    repeat until $X = Y$
    begin
        $X := Y$;
        $Y := Y \cup (W \cap \mathrm{pre}_\exists(Y))$
    end
    return $Y$
end
```

Figure 3.31. The function SAT$_{\mathrm{EU}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying $\mathrm{E}[\phi \mathrm{\ U\ } \psi]$ in the manner described in the labelling algorithm.

The 'state explosion' problem Although the labelling algorithm (with the clever way of handling EG) is linear in the size of the model, unfortunately the size of the model is itself more often than not exponential in the number of variables and the number of components of the system which execute in parallel. This means that, for example, adding a boolean variable to your program will *double* the complexity of verifying a property of it.

The tendency of state spaces to become very large is known as the *state explosion* problem. A lot of research has gone into finding ways of overcoming it, including the use of:

- Efficient data structures, called *ordered binary decision diagrams* (OBDDs), which represent *sets* of states instead of individual states. We study these in Chapter 6 in detail. SMV is implemented using OBDDs.
- Abstraction: one may interpret a model abstractly, uniformly or for a specific property.
- Partial order reduction: for asynchronous systems, several interleavings of component traces may be equivalent as far as satisfaction of the formula to be checked is concerned. This can often substantially reduce the size of the model-checking problem.
- Induction: model-checking systems with (e.g.) large numbers of identical, or similar, components can often be implemented by 'induction' on this number.

- Composition: break the verification problem down into several simpler verification problems.

The last four issues are beyond the scope of this book, but references may be found at the end of this chapter.

3.6.2 CTL model checking with fairness

The verification of $\mathcal{M}, s_0 \vDash \phi$ might fail because the model $\mathcal{M}$ may contain behaviour which is unrealistic, or guaranteed not to occur in the actual system being analysed. For example, in the mutual exclusion case, we expressed that the process `prc` can stay in its critical section (`st=c`) as long as it needs. We modelled this by the non-deterministic assignment

```
next(st) :=
   case
      ...
      (st = c)    : {c,n};
      ...
   esac;
```

However, if we really allow process 2 to stay in its critical section as long as it likes, then we have a path which violates the liveness constraint $\mathrm{AG}\,(t_1 \to \mathrm{AF}\,c_1)$, since, if process 2 stays forever in its critical section, t_1 can be true without c_1 ever becoming true.

We would like to ignore this path, i.e., we would like to assume that the process can stay in its critical section as long as it needs, *but will eventually exit from its critical section* after some finite time.

In LTL, we could handle this by verifying a formula like $\mathrm{FG}\neg c_2 \to \phi$, where ϕ is the formula we actually want to verify. This whole formula asserts that all paths which satisfy infinitely often $\neg c_2$ also satisfy ϕ. However, we cannot do this in CTL because we cannot write formulas of the form $\mathrm{FG}\neg c_2 \to \phi$ in CTL. The logic CTL is not expressive enough to allow us to pick out the 'fair' paths, i.e., those in which process 2 always eventually leaves its critical section.

It is for that reason that SMV allows us to impose *fairness constraints* on top of the transition system it describes. These assumptions state that a given formula is true infinitely often along every computation path. We call such paths *fair computation paths.* The presence of fairness constraints means that, when evaluating the truth of CTL formulas in specifications, the connectives A and E range only over fair paths.

We therefore impose the fairness constraint that `!st=c` be true infinitely often. This means that, whatever state the process is in, there will be a state in the future in which it is not in its critical section. Similar fairness constraints were used for the Alternating Bit Protocol.

Fairness constraints of the form (where ϕ is a state formula)

Property ϕ is true infinitely often

are known as *simple* fairness constraints. Other types include those of the form

If ϕ is true infinitely often, then ψ is also true infinitely often.

SMV can deal only with simple fairness constraints; but how does it do that? To answer that, we now explain how we may adapt our model-checking algorithm so that A and E are assumed to range only over fair computation paths.

Definition 3.21 Let $C \stackrel{\text{def}}{=} \{\psi_1, \psi_2, \ldots, \psi_n\}$ be a set of n fairness constraints. A computation path $s_0 \to s_1 \to \ldots$ is fair with respect to these fairness constraints iff for each i there are infinitely many j such that $s_j \vDash \psi_i$, that is, each ψ_i is true infinitely often along the path. Let us write A_C and E_C for the operators A and E restricted to fair paths.

For example, $\mathcal{M}, s_0 \vDash \mathrm{A}_C\mathrm{G}\,\phi$ iff ϕ is true in every state along all fair paths; and similarly for $\mathrm{A}_C\mathrm{F}$, $\mathrm{A}_C\mathrm{U}$, etc. Notice that these operators explicitly depend on the chosen set C of fairness constraints. We already know that $\mathrm{E}_C\mathrm{U}$, $\mathrm{E}_C\mathrm{G}$ and $\mathrm{E}_C\mathrm{X}$ form an adequate set; this can be shown in the same manner as was done for the temporal connectives without fairness constraints (Section 3.4.4). We also have that

$$\begin{aligned} \mathrm{E}_C[\phi \,\mathrm{U}\, \psi] &\equiv \mathrm{E}[\phi \,\mathrm{U}\, (\psi \wedge \mathrm{E}_C\mathrm{G}\,\top)] \\ \mathrm{E}_C\mathrm{X}\,\phi &\equiv \mathrm{EX}\,(\phi \wedge \mathrm{E}_C\mathrm{G}\,\top). \end{aligned}$$

To see this, observe that a computation path is fair iff any suffix of it is fair. Therefore, we need only provide an algorithm for $\mathrm{E}_C\mathrm{G}\,\phi$. It is similar to Algorithm 2 for EG, given earlier in this chapter:

- Restrict the graph to states satisfying ϕ; of the resulting graph, we want to know from which states there is a fair path.
- Find the maximal *strongly connected components* (SCCs) of the restricted graph;
- Remove an SCC if, for some ψ_i, it does not contain a state satisfying ψ_i. The resulting SCCs are the fair SCCs. Any state of the restricted graph that can reach one has a fair path from it.

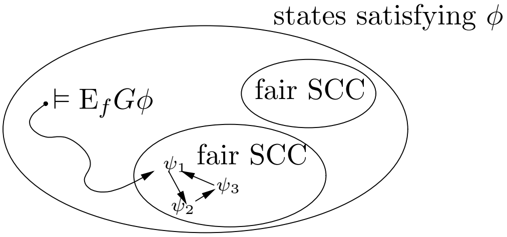


Figure 3.32. Computing the states satisfying $\mathrm{E}_C\mathrm{G}\,\phi$. A state satisfies $\mathrm{E}_C\mathrm{G}\,\phi$ iff, in the graph resulting from the restriction to states satisfying ϕ, the state has a fair path from it. A fair path is one which leads to an SCC with a cycle passing through at least one state that satisfies each fairness constraint; in the example, C equals $\{\psi_1, \psi_2, \psi_3\}$.

- Use backwards breadth-first search to find the states on the restricted graph that can reach a fair SCC.

See Figure 3.32. The complexity of this algorithm is $O(n \cdot f \cdot (V + E))$, i.e., still linear in the size of the model and formula.

It should be noted that writing fairness conditions using SMV's FAIRNESS keyword is necessary only for CTL model checking. In the case of LTL, we can assert the fairness condition as part of the formula to be checked. For example, if we wish to check the LTL formula ψ under the assumption that ϕ is infinitely often true, we check $\mathrm{G\,F}\,\phi \to \psi$. This means: all paths satisfying infinitely often ϕ also satisfy ψ. It is not possible to express this in CTL. In particular, any way of adding As or Es to $\mathrm{G\,F}\,\phi \to \psi$ will result in a formula with a different meaning from the intended one. For example, $\mathrm{AG\,AF}\,\phi \to \psi$ means that if all paths are fair then ψ holds, rather than what was intended: ψ holds along all paths which are fair.

3.6.3 The LTL model-checking algorithm

The algorithm presented in the sections above for CTL model checking is quite intuitive: given a system and a CTL formula, it labels states of the system with the subformulas of the formula which are satisfied there. The state-labelling approach is appropriate because subformulas of the formula may be evaluated in states of the system. This is not the case for LTL: subformulas of the formula must be evaluated not in states but *along paths* of the system. Therefore, LTL model checking has to adopt a different strategy.

There are several algorithms for LTL model checking described in the literature. Although they differ in detail, nearly all of them adopt the same basic strategy. We explain that strategy first; then, we describe some algorithms in more detail.

The basic strategy Let $\mathcal{M} = (S, \rightarrow, L)$ be a model, $s \in S$, and ϕ an LTL formula. We determine whether $\mathcal{M}, s \vDash \phi$, i.e., whether ϕ is satisfied along all paths of $\mathcal{M}$ starting at s. Almost all LTL model checking algorithms proceed along the following three steps.

1. Construct an automaton, also known as a tableau, for the formula $\neg\phi$. The automaton for ψ is called A_ψ. Thus, we construct $A_{\neg\phi}$. The automaton has a notion of *accepting a trace*. A trace is a sequence of valuations of the propositional atoms. From a path, we can abstract its trace. The construction has the property that for all paths π: $\pi \vDash \psi$ iff the trace of π is accepted by A_ψ. In other words, the automaton A_ψ encodes precisely the traces which satisfy ψ.
   Thus, the automaton $A_{\neg\phi}$ which we construct for $\neg\phi$ has the property that it encodes all the traces satisfying $\neg\phi$; i.e., all the traces which do not satisfy ϕ.
2. Combine the automaton $A_{\neg\phi}$ with the model $\mathcal{M}$ of the system. The combination operation results in a transition system whose paths are *both* paths of the automaton *and* paths of the system.
3. Discover whether there is any path from a state derived from s in the combined transition system. Such a path, if there is one, can be interpreted as a path in $\mathcal{M}$ beginning at s which does not satisfy ϕ.
   If there was *no such path*, then output: 'Yes, $\mathcal{M}, s \vDash \phi$.' Otherwise, if there *is such a path*, output 'No, $\mathcal{M}, s \nvDash \phi$.' In the latter case, the counterexample can be extracted from the path found.

Let us consider an example. The system is described by the SMV program and its model $\mathcal{M}$, shown in Figure 3.33. We consider the formula $\neg(a \text{ U } b)$. Since it is not the case that all paths of $\mathcal{M}$ satisfy the formula (for example, the path $q_3, q_2, q_2 \dots$ does not satisfy it) we expect the model check to fail.

In accordance with Step 1, we construct an automaton $A_{a\text{U}b}$ which characterises precisely the traces which satisfy $a \text{ U } b$. (We use the fact that $\neg\neg(a \text{ U } b)$ is equivalent to $a \text{ U } b$.) Such an automaton is shown in Figure 3.34. We will look at how to construct it later; for now, we just try to understand how and why it works.

A trace t is accepted by an automaton like the one of Figure 3.34 if there exists a path π through the automaton such that:

- π starts in an initial state (i.e. one containing ϕ);
- it respects the transition relation of the automaton;
- t is the trace of π; matches the corresponding state of π;

```
init(a) := 1;
init(b) := 0;
next(a) := case
             !a : 0;
             b  : 1;
             1  : {0,1};
           esac;
next(b) := case
             a & next(a) : !b;
             !a : 1;
             1  : {0,1};
           esac;
```

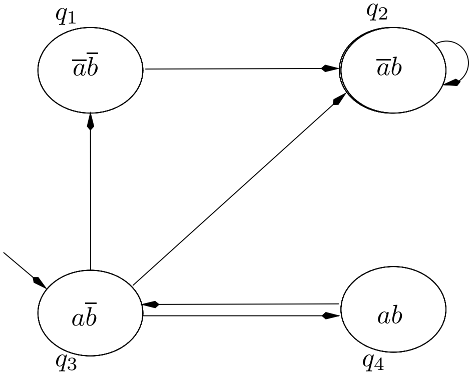


Figure 3.33. An SMV program and its model $\mathcal{M}$.

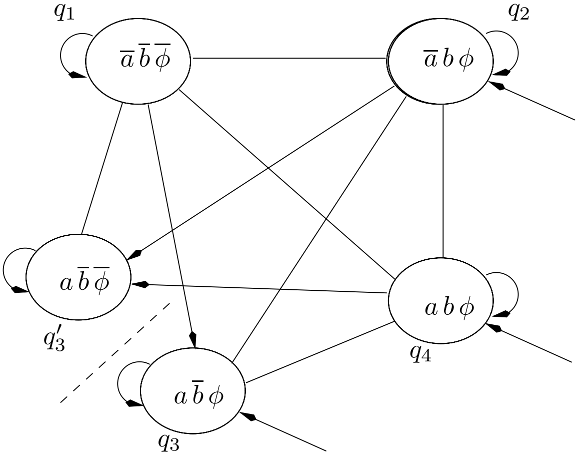


Figure 3.34. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} a \text{ U } b$. The transitions with no arrows can be taken in either direction. The acceptance condition is that the path of the automaton cannot loop indefinitely through q_3.

- the path respects a certain 'accepting condition.' For the automaton of Figure 3.34, the accepting condition is that the path should not end $q_3, q_3, q_3 \dots$, indefinitely.

For example, suppose t is $a\,\overline{b}, a\,\overline{b}, a\,\overline{b}, a\,b, a\,b, \overline{a}\,\overline{b}, a\,\overline{b}, a\,\overline{b}, \dots$, eventually repeating forevermore the state $a\,\overline{b}$. Then we choose the path $q_3, q_3, q_3, q_4, q_4, q_1, q_3', q_3' \dots$. We start in q_3 because the first state is $a\,\overline{b}$ and it is an initial state. The next states we choose just follow the valuation of the states of π. For example, at q_1 the next valuation is $a\,\overline{b}$ and the transitions allow us to choose q_3 or q_3'. We choose q_3', and loop there forevermore. This path meets the conditions, and therefore the trace t is accepted. Observe that the definition states 'there exists a path.' In the example above, there are also paths which don't meet the conditions:

- Any path beginning $q_3, q_3', \ldots$ doesn't meet the condition that we have to respect the transition relation.
- The path $q_3, q_3, q_3, q_4, q_4, q_1, q_3, q_3 \ldots$ doesn't meet the condition that we must not end on a loop of q_3.

These paths need not bother us, because it is sufficient to find one which does meet the conditions in order to declare that π is accepted.

Why does the automaton of Figure 3.34 work as intended? To understand it, observe that it has enough states to distinguish the values of the propositions – that is, a state for each of the valuations $\{\overline{a}\,\overline{b}, \overline{a}\,b, a\,\overline{b}, a\,b\}$, and in fact two states for the valuation $a\,\overline{b}$. One state for each of $\{\overline{a}\,\overline{b}, \overline{a}\,b, a\,b\}$ is intuitively enough, because those valuations determine whether a U b holds. But a U b could be false or true in $a\,\overline{b}$, so we have to consider the two cases. The presence of $\phi \stackrel{\text{def}}{=} a \mathrm{~U~} b$ in a state indicates that either we are still expecting ϕ to become true, or we have just obtained it. Whereas $\overline{\phi}$ indicates we no longer expect ϕ, and have not just obtained it. The transitions of the automaton are such that the only way out of q_3 is to obtain b, i.e., to move to q_2 or q_4. Apart from that, the transitions are liberal, allowing any path to be followed; each of q_1, q_2, q_3 can transition to any valuation, and so can q_3, q_3' taken together, provided we are careful to choose the right one to enter. The acceptance condition, which allows any path except one looping indefinitely on q_3, guarantees that the promise of a U b to deliver b is eventually fulfilled.

Using this automaton $A_{a\mathrm{U}b}$, we proceed to Step 2. To combine the automaton $A_{a\mathrm{U}b}$ with the model of the system $\mathcal{M}$ shown in Figure 3.33, it is convenient first to redraw $\mathcal{M}$ with two versions of q_3; see Figure 3.35(left). It is an equivalent system; all ways into q_3 now non-deterministically choose q_3 or q_3', and which ever one we choose leads to the same successors. But it allows us to superimpose it on $A_{a\mathrm{U}b}$ and select the transitions common to both, obtaining the combined system of Figure 3.35(right).

Step 3 now asks whether there is a path from q of the combined automaton. As can be seen, there are two kinds of path in the combined system: $q_3, (q_4, q_3,)^* q_2, q_2 \ldots,$ and $q_3, q_4, (q_3, q_4,)^* q_3', q_1, q_2, q_2, \ldots$ where $(q_3, q_4)^*$ denotes either the empty string or q_3, q_4 or q_3, q_4, q_3, q_4 etc. Thus, according

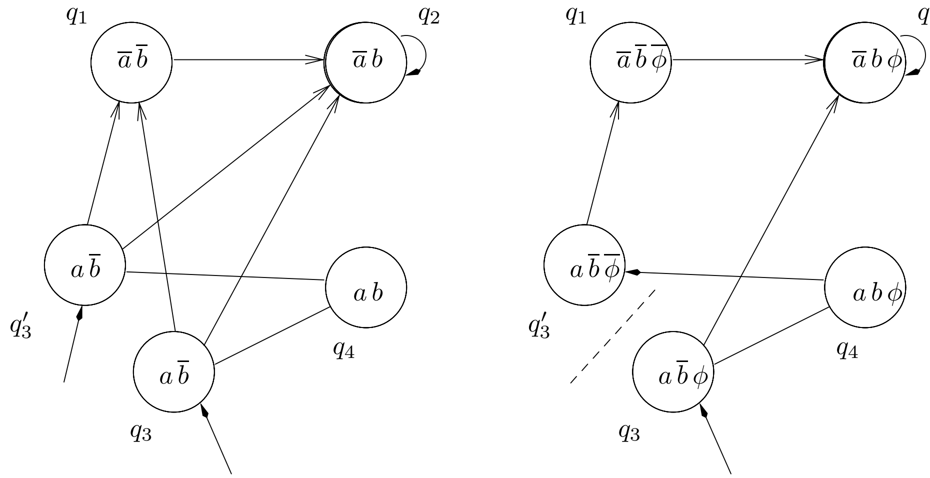


Figure 3.35. Left: the system $\mathcal{M}$ of Figure 3.33, redrawn with an expanded state space; right: the expanded $\mathcal{M}$ and $A_{a\mathrm{U}b}$ combined.

to Step 3, and as we expected, $\neg(a \mathrm{~U~} b)$ is not satisfied in all paths of the original system $\mathcal{M}$.

Constructing the automaton Let us look in more detail at how the automaton is constructed. Given an LTL formula ϕ, we wish to construct an automaton A_ϕ such that A_ϕ accepts precisely those runs on which ϕ holds. We assume that ϕ contains only the temporal connectives U and X; recall that the other temporal connectives can be written in terms of these two.

Define the *closure* $\mathcal{C}(\phi)$ of formula ϕ as the set of subformulas of ϕ and their complements, identifying $\neg\neg\psi$ and ψ. For example, $\mathcal{C}(a \mathrm{~U~} b) = \{a, b, \neg a, \neg b, a \mathrm{~U~} b, \neg(a \mathrm{~U~} b)\}$. The states of A_ϕ, denoted by q, q' etc., are the maximal subsets of $\mathcal{C}(\phi)$ which satisfy the following conditions:

- For all (non-negated) $\psi \in \mathcal{C}(\phi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both.
- $\psi_1 \vee \psi_2 \in q$ holds iff $\psi_1 \in q$ or $\psi_2 \in q$, whenever $\psi_1 \vee \psi_2 \in \mathcal{C}(\phi)$.
- Conditions for other boolean combinations are similar.
- If $\psi_1 \mathrm{~U~} \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\neg(\psi_1 \mathrm{~U~} \psi_2) \in q$, then $\neg\psi_2 \in q$.

Intuitively, these conditions imply that the states of A_ϕ are capable of saying which subformulas of ϕ are true.

The initial states of A_ϕ are those states containing ϕ. For transition relation δ of A_ϕ we have $(q, q') \in \delta$ iff all of the following conditions hold:

- if $\mathrm{X}\,\psi \in q$ then $\psi \in q'$;
- if $\neg\mathrm{X}\,\psi \in q$ then $\neg\psi \in q'$;
- If $\psi_1 \mathrm{~U~} \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1 \mathrm{~U~} \psi_2 \in q'$;
- If $\neg(\psi_1 \mathrm{~U~} \psi_2) \in q$ and $\psi_1 \in q$ then $\neg(\psi_1 \mathrm{~U~} \psi_2) \in q'$.

These last two conditions are justified by the recursion laws

$$\psi_1 \mathrm{~U~} \psi_2 = \psi_2 \vee (\psi_1 \wedge \mathrm{X}\,(\psi_1 \mathrm{~U~} \psi_2))$$
$$\neg(\psi_1 \mathrm{~U~} \psi_2) = \neg\psi_2 \wedge (\neg\psi_1 \vee \mathrm{X}\,\neg(\psi_1 \mathrm{~U~} \psi_2)) \ .$$

In particular, they ensure that whenever some state contains $\psi_1 \mathrm{~U~} \psi_2$, subsequent states contain ψ_1 for as long as they do not contain ψ_2.

As we have defined A_ϕ so far, not all paths through A_ϕ satisfy ϕ. We use additional *acceptance conditions* to guarantee the ‘eventualities’ ψ promised by the formula $\psi_1 \mathrm{~U~} \psi_2$, namely that A_ϕ cannot stay for ever in states satisfying ψ_1 without ever obtaining ψ_2. Recall that, for the automaton of Figure 3.34 for $a \mathrm{~U~} b$, we stipulated the acceptance condition that the path through the automaton should not end $q_3, q_3, \ldots$.

The acceptance conditions of A_ϕ are defined so that they ensure that every state containing some formula $\chi \mathrm{~U~} \psi$ will eventually be followed by some state containing ψ. Let $\chi_1 \mathrm{~U~} \psi_1, \ldots, \chi_k \mathrm{~U~} \psi_k$ be all subformulas of this form in $\mathcal{C}(\phi)$. We stipulate the following acceptance condition: a run is accepted if, for every i such that $1 \leq i \leq k$, the run has infinitely many states satisfying $\neg(\chi_i \mathrm{~U~} \psi_i) \vee \psi_i$. To understand why this condition has the desired effect, imagine the circumstances in which it is false. Suppose we have a run having only finitely many states satisfying $\neg(\chi_i \mathrm{~U~} \psi_i) \vee \psi_i$. Let us advance through all those finitely many states, taking the suffix of the run none of whose states satisfies $\neg(\chi_i \mathrm{~U~} \psi_i) \vee \psi_i$, i.e., all of whose states satisfy $(\chi_i \mathrm{~U~} \psi_i) \wedge \neg\psi_i$. That is precisely the sort of run we want to eliminate.

If we carry out this construction on $a \mathrm{~U~} b$, we obtain the automaton shown in Figure 3.34. Another example is shown in Figure 3.36, for the formula $(p \mathrm{~U~} q) \vee (\neg p \mathrm{~U~} q)$. Since that formula has two U subformulas, there are two sets specified in the acceptance condition, namely, the states satisfying $p \mathrm{~U~} q$ and the states satisfying $\neg p \mathrm{~U~} q$.

How LTL model checking is implemented in NuSMV In the sections above, we described an algorithm for LTL model checking. Given an LTL formula ϕ and a system $\mathcal{M}$ and a state s of $\mathcal{M}$, we may check whether $\mathcal{M}, s \vDash \phi$ holds by constructing the automaton $A_{\neg\phi}$, combining it with $\mathcal{M}$,

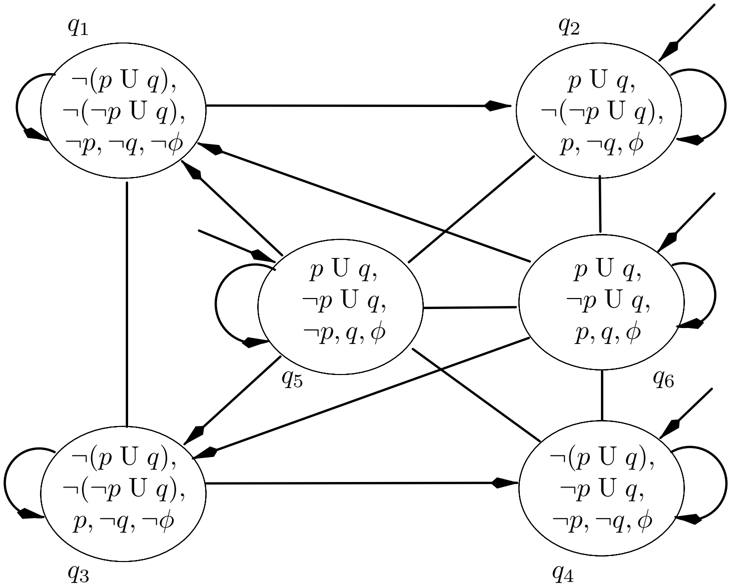

q_1
$\neg(p \text{ U } q)$, $\neg(\neg p \text{ U } q)$, $\neg p, \neg q, \neg\phi$
q_2
$p \text{ U } q$, $\neg(\neg p \text{ U } q)$, $p, \neg q, \phi$
q_5
$p \text{ U } q$, $\neg p \text{ U } q$, $\neg p, q, \phi$
q_6
$p \text{ U } q$, $\neg p \text{ U } q$, p, q, ϕ
q_3
$\neg(p \text{ U } q)$, $\neg(\neg p \text{ U } q)$, $p, \neg q, \neg\phi$
q_4
$\neg(p \text{ U } q)$, $\neg p \text{ U } q$, $\neg p, \neg q, \phi$

Figure 3.36. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} (p \text{ U } q) \vee (\neg p \text{ U } q)$. The transitions with no arrows can be taken in either direction. The acceptance condition asserts that every run must pass infinitely often through the set $\{q_1, q_3, q_4, q_5, q_6\}$, and also the set $\{q_1, q_2, q_3, q_5, q_6\}$.

and checking whether there is a path of the resulting system which satisfies the acceptance condition of $A_{\neg\phi}$.

It is possible to implement the check for such a path in terms of CTL model checking, and this is in fact what NuSMV does. The combined system $\mathcal{M} \times A_{\neg\phi}$ is represented as the system to be model checked in NuSMV, and the formula to be checked is simply $\text{EG}\,\top$. Thus, we ask the question: does the combined system have a path. The acceptance conditions of $A_{\neg\phi}$ are represented as implicit fairness conditions for the CTL model-checking procedure. Explicitly, this amounts to asserting 'FAIRNESS $\neg(\chi \text{ U } \psi) \vee \psi$' for each formula $\chi \text{ U } \psi$ occurring in $\mathcal{C}(\phi)$.

3.7 The fixed-point characterisation of CTL

On page 227, we presented an algorithm which, given a CTL formula ϕ and a model $\mathcal{M} = (S, \rightarrow, L)$, computes the set of states $s \in S$ satisfying ϕ. We write this set as $[\![\phi]\!]$. The algorithm works recursively on the structure of ϕ. For formulas ϕ of height 1 ($\bot$, $\top$ or p), $[\![\phi]\!]$ is computed directly. Other formulas are composed of smaller subformulas combined by a connective of CTL. For example, if ϕ is $\psi_1 \vee \psi_2$, then the algorithm computes the sets $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$ and combines them in a certain way (in this case, by taking the union) in order to obtain $[\![\psi_1 \vee \psi_2]\!]$.

The more interesting cases arise when we deal with a formula such as $\mathrm{EX}\,\psi$, involving a temporal operator. The algorithm computes the set $[\![\psi]\!]$ and then computes the set of all states which have a transition to a state in $[\![\psi]\!]$. This is in accord with the semantics of $\mathrm{EX}\,\psi$: $\mathcal{M}, s \vDash \mathrm{EX}\,\psi$ iff there is a state s' with $s \rightarrow s'$ and $\mathcal{M}, s' \vDash \psi$.

For most of these logical operators, we may easily continue this discussion to see that the algorithms work just as expected. However, the cases EU, AF and EG (where we needed to iterate a certain labelling policy until it stabilised) are not so obvious to reason about. The topic of this section is to develop the semantic insights into these operators that allow us to provide a complete proof for their termination and correctness. Inspecting the pseudocode in Figure 3.28, we see that most of these clauses just do the obvious and correct thing according to the semantics of CTL. For example, try out what SAT does when you call it with $\phi_1 \rightarrow \phi_2$.

Our aim in this section is to prove the termination and correctness of $\mathtt{SAT}_{\mathtt{AF}}$ and $\mathtt{SAT}_{\mathtt{EU}}$. In fact, we will also write a procedure $\mathtt{SAT}_{\mathtt{EG}}$ and prove its termination and correctness[1]. The procedure $\mathtt{SAT}_{\mathtt{EG}}$ is given in Figure 3.37 and is based on the intuitions given in Section 3.6.1: note how *deleting* the label if none of the successor states is labelled is coded as *intersecting* the labelled set with the set of states which have a labelled successor.

The semantics of $\mathrm{EG}\,\phi$ says that $s_0 \vDash \mathrm{EG}\,\phi$ holds iff there exists a computation path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ such that $s_i \vDash \phi$ holds *for all* $i \geq 0$. We could instead express it as follows: $\mathrm{EG}\,\phi$ holds if ϕ holds and $\mathrm{EG}\,\phi$ holds in one of the successor states to the current state. This suggests the equivalence $\mathrm{EG}\,\phi \equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi$ which can easily be proved from the semantic definitions of the connectives.

Observing that $[\![\mathrm{EX}\,\psi]\!] = \mathrm{pre}_\exists([\![\psi]\!])$ we see that the equivalence above can be written as $[\![\mathrm{EG}\,\phi]\!] = [\![\phi]\!] \cap \mathrm{pre}_\exists([\![\mathrm{EG}\,\phi]\!])$. This does not look like a very promising way of calculating $\mathrm{EG}\,\phi$, because we need to know $\mathrm{EG}\,\phi$ in order to work out the right-hand side. Fortunately, there is a way around this apparent circularity, known as computing fixed points, and that is the subject of this section.

[1] Section 3.6.1 handles $\mathrm{EG}\,\phi$ by translating it into $\neg\mathrm{AF}\,\neg\phi$, but we already noted in Section 3.6.1 that EG could be handled directly.

```
function SAT_EG ($\phi$)
/* determines the set of states satisfying EG $\phi$ */
local var $X, Y$
begin
    $Y$ := SAT ($\phi$);
    $X := \emptyset$;
    repeat until $X = Y$
    begin
        $X := Y$;
        $Y := Y \cap \mathrm{pre}_\exists(Y)$
    end
    return $Y$
end
```

Figure 3.37. The pseudo-code for $\mathtt{SAT}_{\mathrm{EG}}$.

3.7.1 Monotone functions

Definition 3.22 Let S be a set of states and $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ a function on the power set of S.

1. We say that F is monotone iff $X \subseteq Y$ implies $F(X) \subseteq F(Y)$ for all subsets X and Y of S.
2. A subset X of S is called a fixed point of F iff $F(X) = X$.

For an example, let $S \stackrel{\mathrm{def}}{=} \{s_0, s_1\}$ and $F(Y) \stackrel{\mathrm{def}}{=} Y \cup \{s_0\}$ for all subsets Y of S. Since $Y \subseteq Y'$ implies $Y \cup \{s_0\} \subseteq Y' \cup \{s_0\}$, we see that F is monotone. The fixed points of F are all subsets of S containing s_0. Thus, F has two fixed points, the sets $\{s_0\}$ and $\{s_0, s_1\}$. Notice that F has a least $(= \{s_0\})$ and a greatest $(= \{s_0, s_1\})$ fixed point.

An example of a function $G\colon \mathcal{P}(S) \to \mathcal{P}(S)$, which is *not* monotone, is given by

$$G(Y) \stackrel{\mathrm{def}}{=} \mathtt{if}\ Y = \{s_0\}\ \mathtt{then}\ \{s_1\}\ \mathtt{else}\ \{s_0\}.$$

So G maps $\{s_0\}$ to $\{s_1\}$ and *all other sets* to $\{s_0\}$. The function G is not monotone since $\{s_0\} \subseteq \{s_0, s_1\}$ but $G(\{s_0\}) = \{s_1\}$ is *not* a subset of $G(\{s_0, s_1\}) = \{s_0\}$. Note that G has *no* fixed points whatsoever.

The reasons for exploring monotone functions on $\mathcal{P}(S)$ in the context of proving the correctness of SAT are:

1. that monotone functions *always* have a least and a greatest fixed point;
2. that the meanings of EG, AF and EU can be expressed via greatest, respectively least, fixed points of monotone functions on $\mathcal{P}(S)$;

3. that these fixed-points can be easily computed, and;
4. that the procedures $\mathtt{SAT}_{\mathtt{EU}}$ and $\mathtt{SAT}_{\mathtt{AF}}$ code up such fixed-point computations, and are correct by item 2.

Notation 3.23 $F^i(X)$ means

$$\underbrace{F(F(\dots F}_{i \text{ times}}(X)\dots))$$

Thus, the function F^i is just 'F applied i many times.'

For example, for the function $F(Y) \stackrel{\text{def}}{=} Y \cup \{s_0\}$, we obtain $F^2(Y) = F(F(Y)) = (Y \cup \{s_0\}) \cup \{s_0\} = Y \cup \{s_0\} = F(Y)$. In this case, $F^2 = F$ and therefore $F^i = F$ for all $i \geq 1$. It is not always the case that the sequence of functions $(F^1, F^2, F^3, \dots)$ stabilises in such a way. For example, this won't happen for the function G defined above (see Exercise 1(d) on page 253). The following fact is a special case of a fundamental insight, often referred to as the Knaster–Tarski Theorem.

Theorem 3.24 Let S be a set $\{s_0, s_1, \dots, s_n\}$ with $n+1$ elements. If $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ is a monotone function, then $F^{n+1}(\emptyset)$ is the least fixed point of F and $F^{n+1}(S)$ is the greatest fixed point of F.

PROOF: Since $\emptyset \subseteq F(\emptyset)$, we get $F(\emptyset) \subseteq F(F(\emptyset))$, i.e., $F^1(\emptyset) \subseteq F^2(\emptyset)$, for F is monotone. We can now use mathematical induction to show that

$$F^1(\emptyset) \subseteq F^2(\emptyset) \subseteq F^3(\emptyset) \subseteq \dots \subseteq F^i(\emptyset)$$

for all $i \geq 1$. In particular, taking $i \stackrel{\text{def}}{=} n+1$, we claim that one of the expressions $F^k(\emptyset)$ above is already a fixed point of F. Otherwise, $F^1(\emptyset)$ needs to contain at least one element (for then $\emptyset \neq F(\emptyset)$). By the same token, $F^2(\emptyset)$ needs to have at least two elements since it must be bigger than $F^1(\emptyset)$. Continuing this argument, we see that $F^{n+2}(\emptyset)$ would have to contain at least $n+2$ many elements. The latter is impossible since S has only $n+1$ elements. Therefore, $F(F^k(\emptyset)) = F^k(\emptyset)$ for some $0 \leq k \leq n+1$, which readily implies that $F^{n+1}(\emptyset)$ is a fixed point of F as well.

Now suppose that X is another fixed point of F. We need to show that $F^{n+1}(\emptyset)$ is a subset of X; but, since $\emptyset \subseteq X$, we conclude $F(\emptyset) \subseteq F(X) = X$, for F is monotone and X a fixed point of F. By induction, we obtain $F^i(\emptyset) \subseteq X$ for all $i \geq 0$. So, for $i \stackrel{\text{def}}{=} n+1$, we get $F^{n+1}(\emptyset) \subseteq X$.

The proof of the statements about the greatest fixed point is dual to the one above. Simply replace $\subseteq$ by $\supseteq$, $\emptyset$ by S and 'bigger' by 'smaller.' □

This theorem about the existence of least and greatest fixed points of monotone functions $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ not only asserted the existence of such fixed points; it also provided a recipe for computing them, and correctly so. For example, in computing the least fixed point of F, all we have to do is apply F to the empty set $\emptyset$ and keep applying F to the result until the latter becomes invariant under the application of F. The theorem above further ensures that this process is *guaranteed to terminate.* Moreover, we can specify an upper bound $n+1$ to the worst-case number of iterations necessary for reaching this fixed point, assuming that S has $n+1$ elements.

3.7.2 The correctness of $\mathtt{SAT}_{\mathtt{EG}}$

We saw at the end of the last section that $[\![\mathrm{EG}\,\phi]\!] = [\![\phi]\!] \cap \mathrm{pre}_\exists([\![\mathrm{EG}\,\phi]\!])$. This implies that $\mathrm{EG}\,\phi$ is a fixed point of the function $F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X)$. In fact, F is monotone, $\mathrm{EG}\,\phi$ is its greatest fixed point and therefore $\mathrm{EG}\,\phi$ can be computed using Theorem 3.24.

Theorem 3.25 Let F be as defined above and let S have $n+1$ elements. Then F is monotone, $[\![\mathrm{EG}\,\phi]\!]$ is the greatest fixed point of F, and $[\![\mathrm{EG}\,\phi]\!] = F^{n+1}(S)$.

PROOF:

1. In order to show that F is monotone, we take any two subsets X and Y of S such that $X \subseteq Y$ and we need to show that $F(X)$ is a subset of $F(Y)$. Given s_0 such that there is some $s_1 \in X$ with $s_0 \to s_1$, we certainly have $s_0 \to s_1$, where $s_1 \in Y$, for X is a subset of Y. Thus, we showed $\mathrm{pre}_\exists(X) \subseteq \mathrm{pre}_\exists(Y)$ from which we readily conclude that $F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X) \subseteq [\![\phi]\!] \cap \mathrm{pre}_\exists(Y) = F(Y)$.
2. We have already seen that $[\![\mathrm{EG}\,\phi]\!]$ is a fixed point of F. To show that it is the greatest fixed point, it suffices to show here that any set X with $F(X) = X$ has to be contained in $[\![\mathrm{EG}\,\phi]\!]$. So let s_0 be an element of such a fixed point X. We need to show that s_0 is in $[\![\mathrm{EG}\,\phi]\!]$ as well. For that we use the fact that
$$s_0 \in X = F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X)$$
to infer that $s_0 \in [\![\phi]\!]$ and $s_0 \to s_1$ for some $s_1 \in X$; but, since s_1 is in X, we may apply that same argument to $s_1 \in X = F(X) = [\![\phi]\!] \cap \mathrm{pre}_\exists(X)$ and we get $s_1 \in [\![\phi]\!]$ and $s_1 \to s_2$ for some $s_2 \in X$. By mathematical induction, we can therefore construct an infinite path $s_0 \to s_1 \to \cdots \to s_n \to s_{n+1} \to \ldots$ such that $s_i \in [\![\phi]\!]$ for all $i \geq 0$. By the definition of $[\![\mathrm{EG}\,\phi]\!]$, this entails $s_0 \in [\![\mathrm{EG}\,\phi]\!]$.
3. The last item is now immediately accessible from the previous one and Theorem 3.24. □

Now we can see that the procedure $\mathtt{SAT}_{\mathtt{EG}}$ is correctly coded and terminates. First, note that the line $Y := Y \cap \mathrm{pre}_{\exists}(Y)$ in the procedure $\mathtt{SAT}_{\mathtt{EG}}$ (Figure 3.37) could be changed to $Y := \mathtt{SAT}(\phi) \cap \mathrm{pre}_{\exists}(Y)$ without changing the effect of the procedure. To see this, note that the first time round the loop, Y *is* $\mathtt{SAT}(\phi)$; and in subsequent loops, $Y \subseteq \mathtt{SAT}(\phi)$, so it doesn't matter whether we intersect with Y or $\mathtt{SAT}(\phi)$[2]. With the change, it is clear that $\mathtt{SAT}_{\mathtt{EG}}$ is calculating the greatest fixed point of F; therefore its correctness follows from Theorem 3.25.

3.7.3 The correctness of $\mathtt{SAT}_{\mathtt{EU}}$

Proving the correctness of $\mathtt{SAT}_{\mathtt{EU}}$ is similar. We start by noting the equivalence $\mathrm{E}[\phi \mathrel{\mathrm{U}} \psi] \equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi \mathrel{\mathrm{U}} \psi])$ and we write it as $\llbracket \mathrm{E}[\phi \mathrel{\mathrm{U}} \psi] \rrbracket = \llbracket \psi \rrbracket \cup (\llbracket \phi \rrbracket \cap \mathrm{pre}_{\exists} \llbracket \mathrm{E}[\phi \mathrel{\mathrm{U}} \psi] \rrbracket)$. That tells us that $\llbracket \mathrm{E}[\phi \mathrel{\mathrm{U}} \psi] \rrbracket$ is a fixed point of the function $G(X) = \llbracket \psi \rrbracket \cup (\llbracket \phi \rrbracket \cap \mathrm{pre}_{\exists}(X))$. As before, we can prove that this function is monotone. It turns out that $\llbracket \mathrm{E}[\phi \mathrel{\mathrm{U}} \psi] \rrbracket$ is its *least* fixed point and that the function $\mathtt{SAT}_{\mathtt{EU}}$ is actually computing it in the manner of Theorem 3.24.

Theorem 3.26 Let G be defined as above and let S have $n+1$ elements. Then G is monotone, $\llbracket \mathrm{E}(\phi \mathrel{\mathrm{U}} \psi) \rrbracket$ is the least fixed point of G, and we have $\llbracket \mathrm{E}(\phi \mathrel{\mathrm{U}} \psi) \rrbracket = G^{n+1}(\emptyset)$.

[2] If you are sceptical, try computing the values $Y_0, Y_1, Y_2, \dots$, where Y_i represents the value of Y after i iterations round the loop. The program before the change computes as follows:

$$\begin{aligned}
Y_0 &= \mathtt{SAT}(\phi)\\
Y_1 &= Y_0 \cap \mathrm{pre}_{\exists}(Y_0)\\
Y_2 &= Y_1 \cap \mathrm{pre}_{\exists}(Y_1)\\
&= Y_0 \cap \mathrm{pre}_{\exists}(Y_0) \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0))\\
&= Y_0 \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0)).
\end{aligned}$$

The last of these equalities follows from the monotonicity of $\mathrm{pre}_{\exists}$.

$$\begin{aligned}
Y_3 &= Y_2 \cap \mathrm{pre}_{\exists}(Y_2)\\
&= Y_0 \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0)) \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0)))\\
&= Y_0 \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0))).
\end{aligned}$$

Again the last one follows by monotonicity. Now look at what the program does after the change:

$$\begin{aligned}
Y_0 &= \mathtt{SAT}(\phi)\\
Y_1 &= \mathtt{SAT}(\phi) \cap \mathrm{pre}_{\exists}(Y_0)\\
&= Y_0 \cap \mathrm{pre}_{\exists}(Y_0)\\
Y_2 &= Y_0 \cap \mathrm{pre}_{\exists}(Y_1)\\
Y_3 &= Y_0 \cap \mathrm{pre}_{\exists}(Y_1)\\
&= Y_0 \cap \mathrm{pre}_{\exists}(Y_0 \cap \mathrm{pre}_{\exists}(Y_0)).
\end{aligned}$$

A formal proof would follow by induction on i.

PROOF:

1. Again, we need to show that $X \subseteq Y$ implies $G(X) \subseteq G(Y)$; but that is essentially the same argument as for F, since the function which sends X to $\mathrm{pre}_\exists(X)$ is monotone and all that G now does is to perform the intersection and union of that set with constant sets $[\![\phi]\!]$ and $[\![\psi]\!]$.
2. If S has $n+1$ elements, then the least fixed point of G equals $G^{n+1}(\emptyset)$ by Theorem 3.24. Therefore it suffices to show that this set equals $[\![\mathrm{E}(\phi \mathrm{~U~} \psi)]\!]$. Simply observe what kind of states we obtain by iterating G on the empty set $\emptyset$: $G^1(\emptyset) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists([\![\emptyset]\!])) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \emptyset) = [\![\psi]\!] \cup \emptyset = [\![\psi]\!]$, which are all states $s_0 \in [\![\mathrm{E}(\phi \mathrm{~U~} \psi)]\!]$, where we chose $i = 0$ according to the definition of Until. Now,
$$G^2(\emptyset) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists(G^1(\emptyset)))$$
tells us that the elements of $G^2(\emptyset)$ are all those $s_0 \in [\![\mathrm{E}(\phi \mathrm{~U~} \psi)]\!]$ where we chose $i \leq 1$. By mathematical induction, we see that $G^{k+1}(\emptyset)$ is the set of all states s_0 for which we chose $i \leq k$ to secure $s_0 \in [\![\mathrm{E}(\phi \mathrm{~U~} \psi)]\!]$. Since this holds for all k, we see that $[\![\mathrm{E}(\phi \mathrm{~U~} \psi)]\!]$ is nothing but the union of all sets $G^{k+1}(\emptyset)$ with $k \geq 0$; but, since $G^{n+1}(\emptyset)$ is a fixed point of G, we see that this union is just $G^{n+1}(\emptyset)$. □

The correctness of the coding of $\mathtt{SAT_{EU}}$ follows similarly to that of $\mathtt{SAT_{EG}}$. We change the line $Y := Y \cup (W \cap \mathrm{pre}_\exists(Y))$ into $Y := \mathtt{SAT}(\psi) \cup (W \cap \mathrm{pre}_\exists(Y))$ and observe that this does not change the result of the procedure, because the first time round the loop, Y is $\mathtt{SAT}(\psi)$; and, since Y is always increasing, it makes no difference whether we perform a union with Y or with $\mathtt{SAT}(\psi)$. Having made that change, it is then clear that $\mathtt{SAT_{EU}}$ is just computing the least fixed point of G using Theorem 3.24.

We illustrate these results about the functions F and G above through an example. Consider the system in Figure 3.38. We begin by computing the set $[\![\mathrm{EF}\, p]\!]$. By the definition of EF this is just $[\![\mathrm{E}(\top \mathrm{~U~} p)]\!]$. So we have $\phi_1 \stackrel{\mathrm{def}}{=} \top$ and $\phi_2 \stackrel{\mathrm{def}}{=} p$. From Figure 3.38, we obtain $[\![p]\!] = \{s_3\}$ and of course $[\![\top]\!] = S$. Thus, the function G above equals $G(X) = \{s_3\} \cup \mathrm{pre}_\exists(X)$. Since $[\![\mathrm{E}(\top \mathrm{~U~} p)]\!]$ equals the least fixed point of G, we need to iterate G on $\emptyset$ until this process stabilises. First, $G^1(\emptyset) = \{s_3\} \cup \mathrm{pre}_\exists(\emptyset) = \{s_3\}$. Second, $G^2(\emptyset) = G(G^1(\emptyset)) = \{s_3\} \cup \mathrm{pre}_\exists(\{s_3\}) = \{s_1, s_3\}$. Third, $G^3(\emptyset) = G(G^2(\emptyset)) = \{s_3\} \cup \mathrm{pre}_\exists(\{s_1, s_3\}) = \{s_0, s_1, s_2, s_3\}$. Fourth, $G^4(\emptyset) = G(G^3(\emptyset)) = \{s_3\} \cup \mathrm{pre}_\exists(\{s_0, s_1, s_2, s_3\}) = \{s_0, s_1, s_2, s_3\}$. Therefore, $\{s_0, s_1, s_2, s_3\}$ is the least fixed point of G, which equals $[\![\mathrm{E}(\top \mathrm{~U~} p)]\!]$ by Theorem 3.20. But then $[\![\mathrm{E}(\top \mathrm{~U~} p)]\!] = [\![\mathrm{EF}\, p]\!]$.

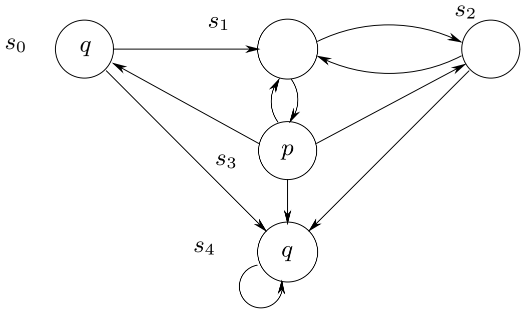


Figure 3.38. A system for which we compute invariants.

The other example we study is the computation of the set $[\![\mathrm{EG}\, q]\!]$. By Theorem 3.25, that set is the greatest fixed point of the function F above, where $\phi \stackrel{\mathrm{def}}{=} q$. From Figure 3.38 we see that $[\![q]\!] = \{s_0, s_4\}$ and so $F(X) = [\![q]\!] \cap \mathrm{pre}_\exists(X) = \{s_0, s_4\} \cap \mathrm{pre}_\exists(X)$. Since $[\![\mathrm{EG}\, q]\!]$ equals the greatest fixed point of F, we need to iterate F on S until this process stabilises. First, $F^1(S) = \{s_0, s_4\} \cap \mathrm{pre}_\exists(S) = \{s_0, s_4\} \cap S$ since every s has some s' with $s \to s'$. Thus, $F^1(S) = \{s_0, s_4\}$.

Second, $F^2(S) = F(F^1(S)) = \{s_0, s_4\} \cap \mathrm{pre}_\exists(\{s_0, s_4\}) = \{s_0, s_4\}$. Therefore, $\{s_0, s_4\}$ is the greatest fixed point of F, which equals $[\![\mathrm{EG}\, q]\!]$ by Theorem 3.25.

3.8 Exercises

Exercises 3.1

1. Read Section 2.7 in case you have not yet done so and classify Alloy and its constraint analyser according to the classification criteria for formal methods proposed on page 172.
2. Visit and browse the websites[3] and[4] to find formal methods that interest you for whatever reason. Then classify them according to the criteria from page 172.

Exercises 3.2

1. Draw parse trees for the LTL formulas:
   (a) $\mathrm{F}\, p \wedge \mathrm{G}\, q \to p \mathrel{\mathrm{W}} r$
   (b) $\mathrm{F}\, (p \to \mathrm{G}\, r) \vee \neg q \mathrel{\mathrm{U}} p$
   (c) $p \mathrel{\mathrm{W}} (q \mathrel{\mathrm{W}} r)$
   (d) $\mathrm{G}\,\mathrm{F}\, p \to \mathrm{F}\, (q \vee s)$

[3] `www.afm.sbu.ac.uk`
[4] `www.cs.indiana.edu/formal-methods-education/`

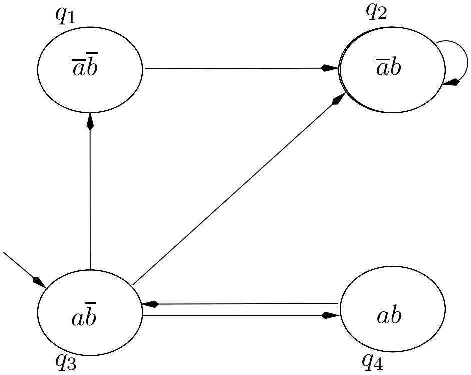


Figure 3.39. A model $\mathcal{M}$.

2. Consider the system of Figure 3.39. For each of the formulas ϕ:
   (a) $\mathrm{G}\, a$
   (b) $a \,\mathrm{U}\, b$
   (c) $a \,\mathrm{U}\, \mathrm{X}\, (a \wedge \neg b)$
   (d) $\mathrm{X}\, \neg b \wedge \mathrm{G}\, (\neg a \vee \neg b)$
   (e) $\mathrm{X}\, (a \wedge b) \wedge \mathrm{F}\, (\neg a \wedge \neg b)$
      (i) Find a path from the initial state q_3 which satisfies ϕ.
      (ii) Determine whether $\mathcal{M}, q_3 \vDash \phi$.
3. Working from the clauses of Definition 3.1 (page 175), prove the equivalences:

$$\begin{aligned}
\phi \,\mathrm{U}\, \psi &\equiv \phi \,\mathrm{W}\, \psi \wedge \mathrm{F}\, \psi \\
\phi \,\mathrm{W}\, \psi &\equiv \phi \,\mathrm{U}\, \psi \vee \mathrm{G}\, \phi \\
\phi \,\mathrm{W}\, \psi &\equiv \psi \,\mathrm{R}\, (\phi \vee \psi) \\
\phi \,\mathrm{R}\, \psi &\equiv \psi \,\mathrm{W}\, (\phi \wedge \psi)\,.
\end{aligned}$$

4. Prove that $\phi \,\mathrm{U}\, \psi \equiv \psi \,\mathrm{R}\, (\phi \vee \psi) \wedge \mathrm{F}\, \psi$.
5. List all subformulas of the LTL formula $\neg p \,\mathrm{U}\, (\mathrm{F}\, r \vee G \neg q \to q \,\mathrm{W}\, \neg r)$.
6. ‘Morally’ there ought to be a dual for W. Work out what it might mean, and then pick a symbol based on the first letter of the meaning.
7. Prove that for all paths π of all models, $\pi \vDash \phi \,\mathrm{W}\, \psi \wedge \mathrm{F}\, \psi$ implies $\pi \vDash \phi \,\mathrm{U}\, \psi$. That is, prove the remaining half of equivalence (3.2) on page 185.
8. Recall the algorithm `NNF` on page 62 which computes the negation normal form of propositional logic formulas. Extend this algorithm to LTL: you need to add program clauses for the additional connectives X, F, G and U, R and W; these clauses have to animate the semantic equivalences that we presented in this section.

Exercises 3.3

1. Consider the model in Figure 3.9 (page 193).
   * (a) Verify that `G(req -> F busy)` holds in all initial states.
   (b) Does ¬(`req` U ¬`busy`) hold in all initial states of that model?
   (c) NuSMV has the capability of referring to the next value of a declared variable `v` by writing `next(v)`. Consider the model obtained from Figure 3.9 by removing the self-loop on state `!req & busy`. Use the NuSMV feature `next(...)` to code that modified model as an NuSMV program with the specification `G(req -> F busy)`. Then run it.
2. Verify Remark 3.11 from page 190.

* 3. Draw the transition system described by the ABP program.

Remarks: There are 28 reachable states of the ABP program. (Looking at the program, you can see that the state is described by nine boolean variables, namely `S.st`, `S.message1`, `S.message2`, `R.st`, `R.ack`, `R.expected`, `msg_chan.output1`, `msg_chan.output2` and finally `ack_chan.output`. Therefore, there are $2^9 = 512$ states in total. However, only 28 of them can be reached from the initial state by following a finite path.)

If you abstract away from the contents of the message (e.g., by setting `S.message1` and `msg_chan.output1` to be constant 0), then there are only 12 reachable states. This is what you are asked to draw.

Exercises 3.4

1. Write the parse trees for the following CTL formulas:
   * (a) $\mathrm{EG}\, r$
   * (b) $\mathrm{AG}\,(q \to \mathrm{EG}\, r)$
   * (c) $\mathrm{A}[p \;\mathrm{U}\; \mathrm{EF}\, r]$
   * (d) $\mathrm{EF}\,\mathrm{EG}\, p \to \mathrm{AF}\, r$, recall Convention 3.13
   (e) $\mathrm{A}[p \;\mathrm{U}\; \mathrm{A}[q \;\mathrm{U}\; r]]$
   (f) $\mathrm{E}[\mathrm{A}[p \;\mathrm{U}\; q] \;\mathrm{U}\; r]$
   (g) $\mathrm{AG}\,(p \to \mathrm{A}[p \;\mathrm{U}\; (\neg p \wedge \mathrm{A}[\neg p \;\mathrm{U}\; q])])$.
2. Explain why the following are not well-formed CTL formulas:
   * (a) $\mathrm{F}\,\mathrm{G}\, r$
   (b) $\mathrm{X}\,\mathrm{X}\, r$
   (c) $\mathrm{A}\neg\mathrm{G}\,\neg p$
   (d) $\mathrm{F}\,[r \;\mathrm{U}\; q]$
   (e) $\mathrm{EX}\,\mathrm{X}\, r$
   * (f) $\mathrm{AEF}\, r$
   * (g) $\mathrm{AF}\,[(r \;\mathrm{U}\; q) \wedge (p \;\mathrm{U}\; r)]$.
3. State which of the strings below are well-formed CTL formulas. For those which are well-formed, draw the parse tree. For those which are not well-formed, explain why not.

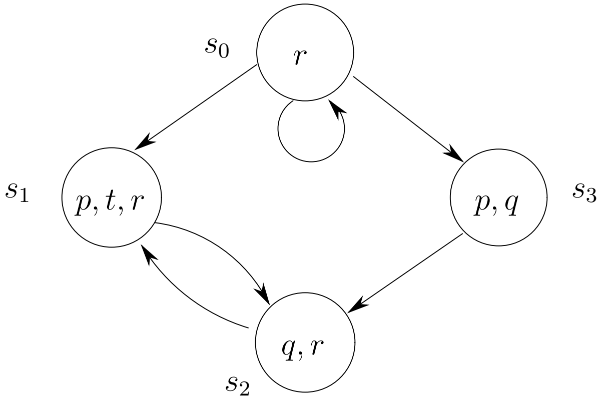


Figure 3.40. A model with four states.

(a) $\neg(\neg p) \vee (r \wedge s)$
(b) $\mathrm{X}\, q$
* (c) $\neg \mathrm{AX}\, q$
(d) $p\ \mathrm{U}\ (\mathrm{AX}\, \bot)$
* (e) $\mathrm{E}[(\mathrm{AX}\, q)\ \mathrm{U}\ (\neg(\neg p) \vee (\top \wedge s))]$
* (f) $(\mathrm{F}\, r) \wedge (\mathrm{AG}\, q)$
(g) $\neg(\mathrm{AG}\, q) \vee (\mathrm{EG}\, q)$.

* 4. List all subformulas of the formula $\mathrm{AG}\,(p \to \mathrm{A}[p\ \mathrm{U}\ (\neg p \wedge \mathrm{A}[\neg p\ \mathrm{U}\ q])])$.

5. Does E[`req` U ¬`busy`] hold in all initial states of the model in Figure 3.9 on page 193?

6. Consider the system $\mathcal{M}$ in Figure 3.40.
(a) Beginning from state s_0, unwind this system into an infinite tree, and draw all computation paths up to length 4 (= the first four layers of that tree).
(b) Determine whether $\mathcal{M}, s_0 \vDash \phi$ and $\mathcal{M}, s_2 \vDash \phi$ hold and justify your answer, where ϕ is the LTL or CTL formula:
* (i) $\neg p \to r$
(ii) $\mathrm{F}\, t$
*(iii) $\neg \mathrm{EG}\ r$
(iv) $\mathrm{E}\,(t\ \mathrm{U}\ q)$
(v) $\mathrm{F}\, q$
(vi) $\mathrm{EF}\ q$
(vii) $\mathrm{EG}\ r$
(viii) $\mathrm{G}\,(r \vee q)$.

7. Let $\mathcal{M} = (S, \to, L)$ be any model for CTL and let $[\![\phi]\!]$ denote the set of all $s \in S$ such that $\mathcal{M}, s \vDash \phi$. Prove the following set identities by inspecting the clauses of Definition 3.15 from page 211.
* (a) $[\![\top]\!] = S$,
(b) $[\![\bot]\!] = \emptyset$

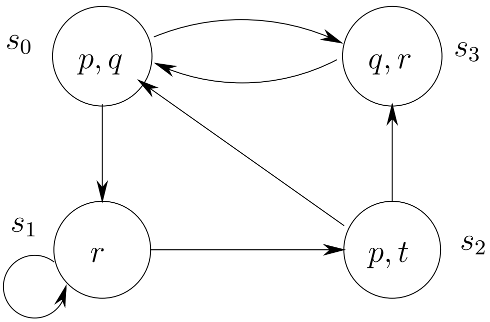


Figure 3.41. Another model with four states.

(c) $[\![\neg\phi]\!] = S - [\![\phi]\!]$,

(d) $[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$

(e) $[\![\phi_1 \vee \phi_2]\!] = [\![\phi_1]\!] \cup [\![\phi_2]\!]$

* (f) $[\![\phi_1 \to \phi_2]\!] = (S - [\![\phi_1]\!]) \cup [\![\phi_2]\!]$

* (g) $[\![\mathrm{AX}\,\phi]\!] = S - [\![\mathrm{EX}\,\neg\phi]\!]$

(h) $[\![\mathrm{A}(\phi_2 \;\mathrm{U}\; \phi_2)]\!] = [\![\neg(\mathrm{E}(\neg\phi_1 \;\mathrm{U}\; (\neg\phi_1 \wedge \neg\phi_2)) \vee \mathrm{EG}\,\neg\phi_2)]\!]$.

8. Consider the model $\mathcal{M}$ in Figure 3.41. Check whether $\mathcal{M}, s_0 \vDash \phi$ and $\mathcal{M}, s_2 \vDash \phi$ hold for the CTL formulas ϕ:

(a) $\mathrm{AF}\,q$

(b) $\mathrm{AG}\,(\mathrm{EF}\,(p \vee r))$

(c) $\mathrm{EX}\,(\mathrm{EX}\,r)$

(d) $\mathrm{AG}\,(\mathrm{AF}\,q)$.

* 9. The meaning of the temporal operators F, G and U in LTL and AU, EU, AG, EG, AF and EF in CTL was defined to be such that 'the present includes the future.' For example, $\mathrm{EF}\,p$ is true for a state if p is true for that state already. Often one would like corresponding operators such that the future excludes the present. Use suitable connectives of the grammar on page 208 to define such (six) modified connectives as derived operators in CTL.

10. Which of the following pairs of CTL formulas are equivalent? For those which are not, exhibit a model of one of the pair which is not a model of the other:

(a) $\mathrm{EF}\,\phi$ and $\mathrm{EG}\,\phi$

* (b) $\mathrm{EF}\,\phi \vee \mathrm{EF}\,\psi$ and $\mathrm{EF}\,(\phi \vee \psi)$

* (c) $\mathrm{AF}\,\phi \vee \mathrm{AF}\,\psi$ and $\mathrm{AF}\,(\phi \vee \psi)$

(d) $\mathrm{AF}\,\neg\phi$ and $\neg\mathrm{EG}\,\phi$

* (e) $\mathrm{EF}\,\neg\phi$ and $\neg\mathrm{AF}\,\phi$

(f) $\mathrm{A}[\phi_1 \;\mathrm{U}\; \mathrm{A}[\phi_2 \;\mathrm{U}\; \phi_3]]$ and $\mathrm{A}[\mathrm{A}[\phi_1 \;\mathrm{U}\; \phi_2] \;\mathrm{U}\; \phi_3]$, hint: it might make it simpler if you think first about models that have just one path

(g) $\top$ and $\mathrm{AG}\,\phi \to \mathrm{EG}\,\phi$

* (h) $\top$ and $\mathrm{EG}\,\phi \to \mathrm{AG}\,\phi$.

11. Find operators to replace the ?, to make the following equivalences:

* (a) $\mathrm{AG}\,(\phi \wedge \psi) \;\equiv\; \mathrm{AG}\,\phi \;?\; \mathrm{AG}\,\psi$

(b) $\mathrm{EF}\,\neg\phi \;\equiv\; \neg??\phi$

12. State explicitly the meaning of the temporal connectives AR etc., as defined on page 217.
13. Prove the equivalences (3.6) on page 216.

* 14. Write pseudo-code for a recursive function TRANSLATE which takes as input an arbitrary CTL formula ϕ and returns as output an equivalent CTL formula ψ whose only operators are among the set $\{\bot, \neg, \wedge, \mathrm{AF}\,, \mathrm{EU}\,, \mathrm{EX}\,\}$.

Exercises 3.5

1. Express the following properties in CTL and LTL whenever possible. If neither is possible, try to express the property in CTL*:

* (a) Whenever p is followed by q (after finitely many steps), then the system enters an 'interval' in which no r occurs until t.

(b) Event p precedes s and t on all computation paths. (You may find it easier to code the negation of that specification first.)

(c) After p, q is never true. (Where this constraint is meant to apply on all computation paths.)

(d) Between the events q and r, event p is never true.

(e) Transitions to states satisfying p occur at most twice.

* (f) Property p is true for every second state along a path.

2. Explain in detail why the LTL and CTL formulas for the practical specification patterns of pages 183 and 215 capture the stated 'informal' properties expressed in plain English.
3. Consider the set of LTL/CTL formulas $\mathcal{F} = \{\mathrm{F}\,p \to \mathrm{F}\,q, \mathrm{AF}\ p \to \mathrm{AF}\ q, \mathrm{AG}\ (p \to \mathrm{AF}\ q)\}$.

(a) Is there a model such that all formulas hold in it?

(b) For each $\phi \in \mathcal{F}$, is there a model such that ϕ is the only formula in $\mathcal{F}$ satisfied in that model?

(c) Find a model in which no formula of $\mathcal{F}$ holds.

4. Consider the CTL formula $\mathrm{AG}\,(p \to \mathrm{AF}\,(s \wedge \mathrm{AX}\,(\mathrm{AF}\,t)))$. Explain what exactly it expresses in terms of the order of occurrence of events p, s and t.
5. Extend the algorithm `NNF` from page 62 which computes the negation normal form of propositional logic formulas to CTL*. Since CTL* is defined in terms of two syntactic categories (state formulas and path formulas), this requires two separate versions of `NNF` which call each other in a way that is reflected by the syntax of CTL* given on page 218.
6. Find a transition system which distinguishes the following pairs of CTL* formulas, i.e., show that they are not equivalent:

(a) $\mathrm{AF}\,\mathrm{G}\,p$ and $\mathrm{AF}\,\mathrm{AG}\,p$

* (b) $\mathrm{AG}\,\mathrm{F}\,p$ and $\mathrm{AG}\,\mathrm{EF}\,p$

(c) $\mathrm{A}[(p\;\mathrm{U}\;r) \vee (q\;\mathrm{U}\;r)]$ and $\mathrm{A}[(p \vee q)\;\mathrm{U}\;r)]$

* (d) $\mathrm{A}[X\, p \vee \mathrm{X}\,\mathrm{X}\, p]$ and $\mathrm{AX}\, p \vee \mathrm{AX}\,\mathrm{AX}\, p$

(e) $\mathrm{E}[\mathrm{G}\,\mathrm{F}\, p]$ and $\mathrm{EG}\,\mathrm{EF}\, p$.

7. The translation from CTL with boolean combinations of path formulas to plain CTL introduced in Section 3.5.1 is not complete. Invent CTL equivalents for:

* (a) $\mathrm{E}[\mathrm{F}\, p \wedge (q \,\mathrm{U}\, r)]$

* (b) $\mathrm{E}[\mathrm{F}\, p \wedge \mathrm{G}\, q]$.

In this way, we have dealt with all formulas of the form $\mathrm{E}[\phi \wedge \psi]$. Formulas of the form $\mathrm{E}[\phi \vee \psi]$ can be rewritten as $\mathrm{E}[\phi] \vee \mathrm{E}[\psi]$ and $\mathrm{A}[\phi]$ can be written $\neg\mathrm{E}[\neg\phi]$. Use this translation to write the following in CTL:

(c) $\mathrm{E}[(p \,\mathrm{U}\, q) \wedge \mathrm{F}\, p]$

* (d) $\mathrm{A}[(p \,\mathrm{U}\, q) \wedge \mathrm{G}\, p]$

* (e) $\mathrm{A}[\mathrm{F}\, p \to \mathrm{F}\, q]$.

8. The aim of this exercise is to demonstrate the expansion given for AW at the end of the last section, i.e., $\mathrm{A}[p \,\mathrm{W}\, q] \equiv \neg\mathrm{E}[\neg q \,\mathrm{U}\, \neg(p \vee q)]$.

(a) Show that the following LTL formulas are valid (i.e., true in any state of any model):

(i) $\neg q \,\mathrm{U}\, (\neg p \wedge \neg q) \to \neg\mathrm{G}\, p$

(ii) $\mathrm{G}\,\neg q \wedge \mathrm{F}\,\neg p \to \neg q \,\mathrm{U}\, (\neg p \wedge \neg q)$.

(b) Expand $\neg((p \,\mathrm{U}\, q) \vee \mathrm{G}\, p)$ using de Morgan rules and the LTL equivalence $\neg(\phi \,\mathrm{U}\, \psi) \equiv (\neg\psi \,\mathrm{U}\, (\neg\phi \wedge \neg\psi)) \vee \neg\mathrm{F}\,\psi$.

(c) Using your expansion and the facts (i) and (ii) above, show $\neg((p \,\mathrm{U}\, q) \vee \mathrm{G}\, p) \equiv \neg q \,\mathrm{U}\, \neg(p \wedge q)$ and hence show that the desired expansion of AW above is correct.

Exercises 3.6

* 1. Verify ϕ_1 to ϕ_4 for the transition system given in Figure 3.11 on page 198. Which of them require the fairness constraints of the SMV program in Figure 3.10?

2. Try to write a CTL formula that enforces non-blocking and no-strict-sequencing at the same time, for the SMV program in Figure 3.10 (page 196).

* 3. Apply the labelling algorithm to check the formulas ϕ_1, ϕ_2, ϕ_3 and ϕ_4 of the mutual exclusion model in Figure 3.7 (page 188).

4. Apply the labelling algorithm to check the formulas ϕ_1, ϕ_2, ϕ_3 and ϕ_4 of the mutual exclusion model in Figure 3.8 (page 191).

5. Prove that (3.8) on page 228 holds in all models. Does your proof require that for every state s there is some state s' with $s \to s'$?

6. Inspecting the definition of the labelling algorithm, explain what happens if you perform it on the formula $p \wedge \neg p$ (in any state, in any model).

7. Modify the pseudo-code for `SAT` on page 227 by writing a special procedure for $\mathrm{AG}\,\psi_1$, without rewriting it in terms of other formulas[5].

[5] Question: will your routine be more like the routine for AF, or more like that for EG on page 224? Why?

* 8. Write the pseudo-code for $\mathtt{SAT}_{\mathrm{EG}}$, based on the description in terms of deleting labels given in Section 3.6.1.

* 9. For mutual exclusion, draw a transition system which forces the two processes to enter their critical section in strict sequence and show that ϕ_4 is false of its initial state.

10. Use the definition of $\vDash$ between states and CTL formulas to explain why $s \vDash \mathrm{AG}\,\mathrm{AF}\,\phi$ means that ϕ is true infinitely often along every path starting at s.

* 11. Show that a CTL formula ϕ is true on infinitely many states of a computation path $s_0 \to s_1 \to s_2 \to \dots$ iff for all $n \geq 0$ there is some $m \geq n$ such that $s_m \vDash \phi$.

12. Run the NuSMV system on some examples. Try commenting out, or deleting, some of the fairness constraints, if applicable, and see the counter examples NuSMV generates. NuSMV is very easy to run.

13. In the one-bit channel, there are two fairness constraints. We could have written this as a single one, inserting '&' between `running` and the long formula, or we could have separated the long formula into two and made it into a total of three fairness constraints.

    In general, what is the difference between the single fairness constraint $\phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_n$ and the n fairness constraints ϕ_1, $\phi_2, \dots, \phi_n$? Write an SMV program with a fairness constraint `a & b` which is not equivalent to the two fairness constraints `a` and `b`. (You can actually do it in four lines of SMV.)

14. Explain the construction of formula ϕ_4, used to express that the processes need not enter their critical section in strict sequence. Does it rely on the fact that the safety property ϕ_1 holds?

* 15. Compute the $\mathrm{E}_C\mathrm{G}\,\top$ labels for Figure 3.11, given the fairness constraints of the code in Figure 3.10 on page 196.

Exercises 3.7

1. Consider the functions

$$H_1, H_2, H_3 \colon \mathcal{P}(\{1,2,3,4,5,6,7,8,9,10\}) \to \mathcal{P}(\{1,2,3,4,5,6,7,8,9,10\})$$

defined by

$$\begin{aligned} H_1(Y) &\overset{\mathrm{def}}{=} Y - \{1,4,7\} \\ H_2(Y) &\overset{\mathrm{def}}{=} \{2,5,9\} - Y \\ H_3(Y) &\overset{\mathrm{def}}{=} \{1,2,3,4,5\} \cap (\{2,4,8\} \cup Y) \end{aligned}$$

for all $Y \subseteq \{1,2,3,4,5,6,7,8,9,10\}$.

* (a) Which of these functions are monotone; which ones aren't? Justify your answer in each case.

* (b) Compute the least and greatest fixed points of H_3 using the iterations H_3^i with $i = 1, 2, \dots$ and Theorem 3.24.

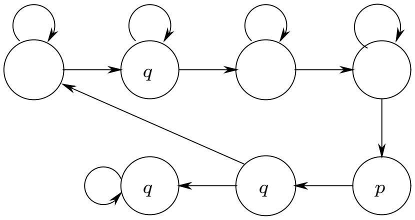


Figure 3.42. Another system for which we compute invariants.

(c) Does H_2 have any fixed points?

(d) Recall $G\colon \mathcal{P}(\{s_0, s_1\}) \to \mathcal{P}(\{s_0, s_1\})$ with

$$G(Y) \stackrel{\text{def}}{=} \texttt{if } Y = \{s_0\} \texttt{ then } \{s_1\} \texttt{ else } \{s_0\}\,.$$

Use mathematical induction to show that G^i equals G for all odd numbers $i \geq 1$. What does G^i look like for even numbers i?

* 2. Let A and B be two subsets of S and let $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ be a monotone function. Show that:

(a) $F_1 : \mathcal{P}(S) \to \mathcal{P}(S)$ with $F_1(Y) \stackrel{\text{def}}{=} A \cap F(Y)$ is monotone;

(b) $F_2 : \mathcal{P}(S) \to \mathcal{P}(S)$ with $F_2(Y) \stackrel{\text{def}}{=} A \cup (B \cap F(Y))$ is monotone.

3. Use Theorems 3.25 and 3.26 to compute the following sets (the underlying model is in Figure 3.42):

(a) $[\![\mathrm{EF}\, p]\!]$

(b) $[\![\mathrm{EG}\, q]\!]$.

4. Using the function $F(X) = [\![\phi]\!] \cup \mathrm{pre}_\forall(X)$ prove that $[\![\mathrm{AF}\,\phi]\!]$ is the least fixed point of F. Hence argue that the procedure $\mathtt{SAT}_{\mathtt{AF}}$ is correct and terminates.

* 5. One may also compute $\mathrm{AG}\,\phi$ directly as a fixed point. Consider the function $H\colon \mathcal{P}(S) \to \mathcal{P}(S)$ with $H(X) = [\![\phi]\!] \cap \mathrm{pre}_\forall(X)$. Show that H is monotone and that $[\![\mathrm{AG}\,\phi]\!]$ is the greatest fixed point of H. Use that insight to write a procedure $\mathtt{SAT}_{\mathtt{AG}}$.

6. Similarly, one may compute $\mathrm{A}[\phi_1 \;\mathrm{U}\; \phi_2]$ directly as a fixed point, using $K\colon \mathcal{P}(S) \to \mathcal{P}(S)$, where $K(X) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathrm{pre}_\forall(X))$. Show that K is monotone and that $[\![\mathrm{A}[\phi_1 \;\mathrm{U}\; \phi_2]]\!]$ is the least fixed point of K. Use that insight to write a procedure $\mathtt{SAT}_{\mathtt{AU}}$. Can you use that routine to handle all calls of the form $\mathrm{AF}\,\phi$ as well?

7. Prove that $[\![\mathrm{A}[\phi_1 \;\mathrm{U}\; \phi_2]]\!] = [\![\phi_2 \vee (\phi_1 \wedge \mathrm{AX}\,(\mathrm{A}[\phi_1 \;\mathrm{U}\; \phi_2]))]\!]$.

8. Prove that $[\![\mathrm{AG}\,\phi]\!] = [\![\phi \wedge \mathrm{AX}\,(\mathrm{AG}\,\phi)]\!]$.

9. Show that the repeat-statements in the code for $\mathtt{SAT}_{\mathtt{EU}}$ and $\mathtt{SAT}_{\mathtt{EG}}$ always terminate. Use this fact to reason informally that the main program SAT terminates for all valid CTL formulas ϕ. Note that some subclauses, like the one for AU, call SAT recursively and with a more complex formula. Why does this not affect termination?

3.9 Bibliographic notes

Temporal logic was invented by the philosopher A. Prior in the 1960s; his logic was similar to what we now call LTL. The first use of temporal logic for reasoning about concurrent programs was by A. Pnueli [Pnu81]. The logic CTL was invented by E. Clarke and E. A. Emerson (during the early 1980s); and CTL* was invented by E. A. Emerson and J. Halpern (in 1986) to unify CTL and LTL.

CTL model checking was invented by E. Clarke and E. A. Emerson [CE81] and by J. Quielle and J. Sifakis [QS81]. The technique we described for LTL model checking was invented by M. Vardi and P. Wolper [VW84]. Surveys of some of these ideas can be found in [CGL93] and [CGP99]. The theorem about adequate sets of CTL connectives is proved in [Mar01].

The original SMV system was written by K. McMillan [McM93] and is available with source code from Carnegie Mellon University[6]. NuSMV[7] is a reimplementation, developed in Trento by A. Cimatti, and M. Roveri and is aimed at being customisable and extensible. Extensive documentation about NuSMV can be found at that site. NuSMV supports essentially the same system description language as CMU SMV, but it has an improved user interface and a greater variety of algorithms. For example, whereas CMU SMV checks only CTL specification, NuSMV supports LTL and CTL. NuSMV implements bounded model checking [BCCZ99]. Cadence SMV[8] is an entirely new model checker focused on compositional systems and abstraction as ways of addressing the state explosion problem. It was also developed by K. McMillan and its description language resembles but much extends the original SMV.

A website which gathers frequently used specification patterns in various frameworks (such as CTL, LTL and regular expressions) is maintained by M. Dwyer, G. Avrunin, J. Corbett and L. Dillon[9].

Current research in model checking includes attempts to exploit abstractions, symmetries and compositionality [CGL94, Lon83, Dam96] in order to reduce the impact of the state explosion problem.

The model checker Spin, which is geared towards asynchronous systems and is based on the temporal logic LTL, can be found at the Spin website[10]. A model checker called FDR2 based on the process algebra CSP is available[11].

[6] www.cs.cmu.edu/~modelcheck/

[7] nusmv.irst.itc.it

[8] www-cad.eecs.berkeley.edu/~kenmcmil/

[9] patterns.projects.cis.ksu.edu/

[10] netlib.bell-labs.com/netlib/spin/whatispin.html

[11] www.fsel.com.fdr2_download.html

The Edinburgh Concurrency Workbench[12] and the Concurrency Workbench of North Carolina[13] are similar software tools for the design and analysis of concurrent systems. An example of a customisable and extensible modular model checking frameworks for the verification of concurrent software is Bogor[14].

There are many textbooks about verification of reactive systems; we mention [MP91, MP95, Ros97, Hol90]. The SMV code contained in this chapter can be downloaded from `www.cs.bham.ac.uk/research/lics/`.

[12] `www.dcs.ed.ac.uk/home/cwb`

[13] `www.cs.sunysb.edu/~cwb`

[14] `http://bogor.projects.cis.ksu.edu/`

4
Program verification

The methods of the previous chapter are suitable for verifying systems of communicating processes, where control is the main issue, but there are no complex *data*. We relied on the fact that those (abstracted) systems are in a *finite state*. These assumptions are not valid for sequential programs running on a single processor, the topic of this chapter. In those cases, the programs may manipulate non-trivial data and – once we admit variables of type integer, list, or tree – we are in the domain of machines with *infinite* state space.

In terms of the classification of verification methods given at the beginning of the last chapter, the methods of this chapter are

Proof-based. We do not exhaustively check every state that the system can get in to, as one does with model checking; this would be impossible, given that program variables can have infinitely many interacting values. Instead, we construct a proof that the system satisfies the property at hand, using a proof calculus. This is analogous to the situation in Chapter 2, where using a suitable proof calculus avoided the problem of having to check infinitely many models of a set of predicate logic formulas in order to establish the validity of a sequent.

Semi-automatic. Although many of the steps involved in proving that a program satisfies its specification are mechanical, there are some steps that involve some intelligence and that cannot be carried out algorithmically by a computer. As we will see, there are often good heuristics to help the programmer complete these tasks. This contrasts with the situation of the last chapter, which was fully automatic.

Property-oriented. Just like in the previous chapter, we verify properties of a program rather than a full specification of its behaviour.

Application domain. The domain of application in this chapter is sequential transformational programs. 'Sequential' means that we assume the program runs on a single processor and that there are no concurrency issues. 'Transformational' means that the program takes an input and, after some computation, is expected to terminate with an output. For example, methods of objects in Java are often programmed in this style. This contrasts with the previous chapter which focuses on reactive systems that are not intended to terminate and that react continually with their environment.

Pre/post-development. The techniques of this chapter should be used during the coding process for small fragments of program that perform an identifiable (and hence, specifiable) task and hence should be used during the development process in order to avoid functional bugs.

4.1 Why should we specify and verify code?

The task of specifying and verifying code is often perceived as an unwelcome addition to the programmer's job and a dispensable one. Arguments in favour of verification include the following:

- **Documentation:** The specification of a program is an important component in its documentation and the process of documenting a program may raise or resolve important issues. The logical structure of the formal specification, written as a formula in a suitable logic, typically serves as a guiding principle in trying to write an implementation in which it holds.
- **Time-to-market:** Debugging big systems during the testing phase is costly and time-consuming and local 'fixes' often introduce new bugs at other places. Experience has shown that verifying programs with respect to formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase and helping in the clarification of the roles and structural aspects of system components.
- **Refactoring:** Properly specified and verified software is easier to reuse, since we have a clear specification of what it is meant to do.
- **Certification audits:** Safety-critical computer systems – such as the control of cooling systems in nuclear power stations, or cockpits of modern aircrafts – demand that their software be specified and verified with as much rigour and formality as possible. Other programs may be commercially critical, such as accountancy software used by banks, and they should be delivered with a warranty: a guarantee for correct performance within proper use. The proof that a program meets its specifications is indeed such a warranty.

The degree to which the software industry accepts the benefits of proper verification of code depends on the perceived extra cost of producing it and the perceived benefits of having it. As verification technology improves, the costs are declining; and as the complexity of software and the extent to which society depends on it increase, the benefits are becoming more important. Thus, we can expect that the importance of verification to industry will continue to increase over the next decades. Microsoft's emergent technology A# combines program verification, testing, and model-checking techniques in an integrated in-house development environment.

Currently, many companies struggle with a legacy of ancient code without proper documentation which has to be adapted to new hardware and network environments, as well as ever-changing requirements. Often, the original programmers who might still remember what certain pieces of code are for have moved, or died. Software systems now often have a longer life-expectancy than humans, which necessitates a durable, transparent and portable design and implementation process; the year-2000 problem was just one such example. Software verification provides some of this.

4.2 A framework for software verification

Suppose you are working for a software company and your task is to write programs which are meant to solve sophisticated problems, or computations. Typically, such a project involves an outside customer – a utility company, for example – who has written up an informal description, in plain English, of the real-world task that is at hand. In this case, it could be the development and maintenance of a database of electricity accounts with all the possible applications of that – automated billing, customer service etc. Since the informality of such descriptions may cause ambiguities which eventually could result in serious and expensive design flaws, it is desirable to condense all the requirements of such a project into formal specifications. These formal specifications are usually symbolic encodings of real-world constraints into some sort of logic. Thus, a framework for producing the software could be:

- Convert the informal description R of requirements for an application domain into an 'equivalent' formula ϕ_R of some symbolic logic;
- Write a program P which is meant to realise ϕ_R in the programming environment supplied by your company, or wanted by the particular customer;
- *Prove* that the program P satisfies the formula ϕ_R.

This scheme is quite crude – for example, constraints may be actual design decisions for interfaces and data types, or the specification may 'evolve' and may partly be ‘unknown’ in big projects – but it serves well as a first approximation to trying to define good programming methodology. Several variations of such a sequence of activities are conceivable. For example, you, as a programmer, might have been given only the formula ϕ_R, so you might have little if any insight into the real-world problem which you are supposed to solve. Technically, this poses no problem, but often it is handy to have both informal and formal descriptions available. Moreover, crafting the informal requirements R is often a mutual process between the client and the programmer, whereby the attempt at formalising R can uncover ambiguities or undesired consequences and hence lead to revisions of R.

This ‘going back and forth’ between the realms of informal and formal specifications is necessary since it is impossible to ‘verify’ whether an *informal* requirement R is equivalent to a *formal* description ϕ_R. The meaning of R as a piece of natural language is grounded in common sense and general knowledge about the real-world domain and often based on heuristics or quantitative reasoning. The meaning of a logic formula ϕ_R, on the other hand, is defined in a precise mathematical, qualitative and compositional way by structural induction on the parse tree of ϕ_R – the first three chapters contain examples of this.

Thus, the process of finding a suitable formalisation ϕ_R of R requires the utmost care; otherwise it is always possible that ϕ_R specifies behaviour which is different from the one described in R. To make matters worse, the requirements R are often inconsistent; customers usually have a fairly vague conception of what exactly a program should do for them. Thus, producing a clear and coherent description R of the requirements for an application domain is already a crucial step in successful programming; this phase ideally is undertaken by customers and project managers around a table, or in a video conference, talking to each other. We address this first item only implicitly in this text, but you should certainly be aware of its importance in practice.

The next phase of the software development framework involves constructing the program P and after that the last task is to verify that P satisfies ϕ_R. Here again, our framework is oversimplifying what goes on in practice, since often proving that P satisfies its specification ϕ_R goes hand-in-hand with inventing a suitable P. This correspondence between proving and programming can be stated quite precisely, but that is beyond the scope of this book.

4.2.1 A core programming language

The programming language which we set out to study here is the typical core language of most imperative programming languages. Modulo trivial syntactic variations, it is a subset of Pascal, C, C++ and Java. Our language consists of assignments to integer- and boolean-valued variables, if-statements, while-statements and sequential compositions. Everything that can be computed by large languages like C and Java can also be computed by our language, though perhaps not as conveniently, because it does not have any objects, procedures, threads or recursive data structures. While this makes it seem unrealistic compared with fully blown commercial languages, it allows us to focus our discussion on the process of formal program verification. The features missing from our language could be implemented on top of it; that is the justification for saying that they do not add to the power of the language, but only to the convenience of using it. Verifying programs using those features would require non-trivial extensions of the proof calculus we present here. In particular, dynamic scoping of variables presents hard problems for program-verification methods, but this is beyond the scope of this book.

Our core language has three syntactic domains: integer expressions, boolean expressions and commands – the latter we consider to be our programs. Integer expressions are built in the familiar way from variables $x, y, z, \dots$, numerals $0, 1, 2, \dots, -1, -2, \dots$ and basic operations like addition $(+)$ and multiplication $(*)$. For example,

$$\begin{array}{l} 5 \\ x \\ 4 + (x - 3) \\ x + (x * (y - (5 + z))) \end{array}$$

are all valid integer expressions. Our grammar for generating integer expressions is

$$E \ ::= \ n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E) \tag{4.1}$$

where n is any numeral in $\{\dots, -2, -1, 0, 1, 2, \dots\}$ and x is any variable. Note that we write multiplication in ‘mathematics’ as $2 \cdot 3$, whereas our core language writes $2 * 3$ instead.

Convention 4.1 In the grammar above, negation $-$ binds more tightly than multiplication $*$, which binds more tightly than subtraction $-$ and addition $+$.

Since if-statements and while-statements contain conditions in them, we also need a syntactic domain B of boolean expressions. The grammar in

Backus Naur form

$$B \ ::= \ \texttt{true} \ \mid \ \texttt{false} \ \mid \ (!B) \ \mid \ (B \,\&\, B) \ \mid \ (B \,||\, B) \ \mid \ (E < E) \qquad (4.2)$$

uses `!` for the negation, `&` for conjunction and `||` for disjunction of boolean expressions. This grammar may be freely expanded by operators which are definable in terms of the above. For example, the test for equality[1] $E_1 == E_2$ may be expressed via $!(E_1 < E_2) \,\&\, !(E_2 < E_1)$. We generally make use of shorthand notation whenever this is convenient. We also write $(E_1 \,!\!= E_2)$ to abbreviate $!(E_1 == E_2)$. We will also assume the usual binding priorities for logical operators stated in Convention 1.3 on page 5. Boolean expressions are built on top of integer expressions since the last clause of (4.2) mentions integer expressions.

Having integer and boolean expressions at hand, we can now define the syntactic domain of commands. Since commands are built from simpler commands using assignments and the control structures, you may think of commands as the actual programs. We choose as grammar for commands

$$C \quad ::= \quad \mathrm{x} = E \ \mid \ C;C \ \mid \ \texttt{if}\ B\ \{C\}\ \texttt{else}\ \{C\} \ \mid \texttt{while}\ B\ \{C\} \qquad (4.3)$$

where the braces { and } are to mark the extent of the blocks of code in the if-statement and the while-statement, as in languages such as C and Java. They can be omitted if the blocks consist of a single statement. The intuitive meaning of the programming constructs is the following:

1. The atomic command `x` $= E$ is the usual assignment statement; it evaluates the integer expression E in the current state of the store and then overwrites the current value stored in x with the result of that evaluation.
2. The compound command $C_1; C_2$ is the sequential composition of the commands C_1 and C_2. It begins by executing C_1 in the current state of the store. If that execution terminates, then it executes C_2 in the storage state resulting from the execution of C_1. Otherwise – if the execution of C_1 does not terminate – the run of $C_1; C_2$ also does not terminate. Sequential composition is an example of a *control structure* since it implements a certain policy of flow of control in a computation.

[1] In common with languages like C and Java, we use a single equals sign = to mean assignment and a double sign == to mean equality. Earlier languages like Pascal used := for assignment and simple = for equality; it is a great pity that C and its successors did not keep this convention. The reason that = is a bad symbol for assignment is that assignment is not symmetric: if we interpret $x = y$ as the assignment, then x becomes y which is not the same thing as y becoming x; yet, $x = y$ and $y = x$ are the same thing if we mean equality. The two dots in := helped remind the reader that this is an asymmetric assignment operation rather than a symmetric assertion of equality. However, the notation = for assignment is now commonplace, so we will use it.

3. Another control structure is `if` B $\{C_1\}$ `else` $\{C_2\}$. It first evaluates the boolean expression B in the current state of the store; if that result is true, then C_1 is executed; if B evaluated to false, then C_2 is executed.
4. The third control construct `while` B $\{C\}$ allows us to write statements which are executed repeatedly. Its meaning is that:

   a the boolean expression B is evaluated in the current state of the store;
   b if B evaluates to false, then the command terminates,
   c otherwise, the command C will be executed. If that execution terminates, then we resume at step (a) with a re-evaluation of B as the updated state of the store may have changed its value.

   The point of the while-statement is that it repeatedly executes the command C as long as B evaluates to true. If B never becomes false, or if one of the executions of C does not terminate, then the while-statement will not terminate. While-statements are the only real source of non-termination in our core programming language.

Example 4.2 The factorial $n!$ of a natural number n is defined inductively by

$$\begin{aligned} 0! &\stackrel{\text{def}}{=} 1 \\ (n+1)! &\stackrel{\text{def}}{=} (n+1)\cdot n! \end{aligned} \tag{4.4}$$

For example, unwinding this definition for n being 4, we get $4! \stackrel{\text{def}}{=} 4 \cdot 3! = \cdots = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 24$. The following program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

is intended to compute the factorial[2] of x and to store the result in y. We will prove that `Fac1` really does this later in the chapter.

4.2.2 Hoare triples

Program fragments generated by (4.3) commence running in a ‘state’ of the machine. After doing some computation, they might terminate. If they do, then the result is another, usually different, state. Since our programming

[2] Please note the difference between the formula $x! = y$, saying that the factorial of x is equal to y, and the piece of code `x != y` which says that `x` is not equal to `y`.

language does not have any procedures or local variables, the 'state' of the machine can be represented simply as a vector of values of all the variables used in the program.

What syntax should we use for ϕ_R, the formal specifications of requirements for such programs? Because we are interested in the output of the program, the language should allow us to talk about the variables in the state after the program has executed, using operators like $=$ to express equality and $<$ for less than. You should be aware of the overloading of $=$. In code, it represents an assignment instruction; in logical formulas, it stands for equality, which we write `==` within program code.

For example, if the informal requirement R says that we should

> Compute a number y whose square is less than the input x.

then an appropriate specification may be $y \cdot y < x$. But what if the input x is -4? There is no number whose square is less than a negative number, so it is not possible to write the program in a way that it will work with all possible inputs. If we go back to the client and say this, he or she is quite likely to respond by saying that the requirement is only that the program work for positive numbers; i.e., he or she *revises* the informal requirement so that it now says

> If the input x is a positive number, compute a number whose square is less than x.

This means we need to be able to talk not just about the state *after* the program executes, but also about the state *before* it executes. The assertions we make will therefore be triples, typically looking like

$$\llparenthesis \phi \rrparenthesis \; P \; \llparenthesis \psi \rrparenthesis \tag{4.5}$$

which (roughly) means:

> If the program P is run in a state that satisfies ϕ, then the state resulting from P's execution will satisfy ψ.

The specification of the program P, to calculate a number whose square is less than x, now looks like this:

$$\llparenthesis x > 0 \rrparenthesis \; P \; \llparenthesis y \cdot y < x \rrparenthesis . \tag{4.6}$$

It means that, if we run P in a state such that $x > 0$, then the resulting state will be such that $y \cdot y < x$. It does not tell us what happens if we run P in a state in which $x \leq 0$, the client required nothing for non-positive values of x. Thus, the programmer is free to do what he or she wants in that case. A program which produces 'garbage' in the case that $x \leq 0$ satisfies the specification, as long as it works correctly for $x > 0$.

Let us make these notions more precise.

Definition 4.3 1. The form $(\!|\phi|\!)\, P\, (\!|\psi|\!)$ of our specification is called a Hoare triple, after the computer scientist C. A. R. Hoare.

2. In (4.5), the formula ϕ is called the precondition of P and ψ is called the postcondition.
3. A store or state of core programs is a function l that assigns to each variable x an integer $l(x)$.
4. For a formula ϕ of predicate logic with function symbols $-$ (unary), $+$, $-$, and $*$ (binary); and a binary predicate symbols $<$ and $=$, we say that a state l satisfies ϕ or l is a ϕ-state – written $l \vDash \phi$ – iff $\mathcal{M} \vDash_l \phi$ from page 128 holds, where l is viewed as a look-up table and the model $\mathcal{M}$ has as set A all integers and interprets the function and predicate symbols in their standard manner.
5. For Hoare triples in (4.5), we demand that quantifiers in ϕ and ψ only bind variables that do not occur in the program P.

Example 4.4 For any state l for which $l(x) = -2$, $l(y) = 5$, and $l(z) = -1$, the relation

1. $l \vDash \neg(x + y < z)$ holds since $x + y$ evaluates to $-2 + 5 = 3$, z evaluates to $l(z) = -1$, and 3 is not strictly less than -1;
2. $l \vDash y - x * z < z$ does not hold, since the lefthand expression evaluates to $5 - (-2) \cdot (-1) = 3$ which is not strictly less than $l(z) = -1$;
3. $l \vDash \forall u\, (y < u \to y * z < u * z)$ does not hold; for u being 7, $l \vDash y < u$ holds, but $l \vDash y * z < u * z$ does not.

Often, we do not want to put any constraints on the initial state; we simply wish to say that, no matter what state we start the program in, the resulting state should satisfy ψ. In that case the precondition can be set to $\top$, which is – as in previous chapters – a formula which is true in any state.

Note that the triple in (4.6) does not specify a unique program P, or a unique behaviour. For example, the program which simply does `y = 0;` satisfies the specification – since $0 \cdot 0$ is less than any positive number – as does the program

```
y = 0;
while (y * y < x) {
   y = y + 1;
   }
y = y - 1;
```

This program finds the greatest y whose square is less than x; the while-statement overshoots a bit, but then we fix it after the while-statement.[3]

[3] We could avoid this inelegance by using the `repeat` construct of exercise 3 on page 299.

Note that these two programs have different behaviour. For example, if x is 22, the first one will compute $y = 0$ and the second will render $y = 4$; but both of them satisfy the specification.

Our agenda, then, is to develop a notion of proof which allows us to prove that a program P satisfies the specification given by a precondition ϕ and a postcondition ψ in (4.5). Recall that we developed proof calculi for propositional and predicate logic where such proofs could be accomplished by investigating the structure of the formula one wanted to prove. For example, for proving an implication $\phi \to \psi$ one had to assume ϕ and manage to show ψ; then the proof could be finished with the proof rule for implies-introduction. The proof calculi which we are about to develop follow similar lines. Yet, they are different from the logics we previously studied since they prove triples which are built from two different sorts of things: logical formulas ϕ and ψ versus a piece of code P. Our proof calculi have to address each of these appropriately. Nonetheless, we retain proof strategies which are *compositional*, but now in the structure of P. Note that this is an important advantage in the verification of big projects, where code is built from a multitude of modules such that the correctness of certain parts will depend on the correctness of certain others. Thus, your code might call subroutines which other members of your project are about to code, but you can already check the correctness of your code by assuming that the subroutines meet their own specifications. We will explore this topic in Section 4.5.

4.2.3 Partial and total correctness

Our explanation of when the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds was rather informal. In particular, it did not say what we should conclude if P does not terminate. In fact there are two ways of handling this situation. *Partial correctness* means that we do not require the program to terminate, whereas in *total correctness* we insist upon its termination.

Definition 4.5 (Partial correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under partial correctness if, for all states which satisfy ϕ, the state resulting from P's execution satisfies the postcondition ψ, provided that P actually terminates. In this case, the relation $\vDash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds. We call $\vDash_{\mathsf{par}}$ the satisfaction relation for partial correctness.

Thus, we insist on ψ being true of the resulting state only if the program P has terminated on an input satisfying ϕ. Partial correctness is rather a weak requirement, since any program which does not terminate at all satisfies its specification. In particular, the program

```
    while true { x = 0; }
```

– which endlessly 'loops' and never terminates – satisfies all specifications, since partial correctness only says what must happen *if* the program terminates.

Total correctness, on the other hand, requires that the program terminates in order for it to satisfy a specification.

Definition 4.6 (Total correctness) We say that the triple $\llparenthesis \phi \rrparenthesis\, P\, \llparenthesis \psi \rrparenthesis$ is satisfied under total correctness if, for all states in which P is executed which satisfy the precondition ϕ, P is guaranteed to terminate and the resulting state satisfies the postcondition ψ. In this case, we say that $\vDash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis\, P\, \llparenthesis \psi \rrparenthesis$ holds and call $\vDash_{\mathsf{tot}}$ the satisfaction relation of total correctness.

A program which 'loops' forever on all input does not satisfy any specification under total correctness. Clearly, total correctness is more useful than partial correctness, so the reader may wonder why partial correctness is introduced at all. Proving total correctness usually benefits from proving partial correctness first and then proving termination. So, although our primary interest is in proving total correctness, it often happens that we have to or may wish to split this into separate proofs of partial correctness and of termination. Most of this chapter is devoted to the proof of partial correctness, though we return to the issue of termination in Section 4.4.

Before we delve into the issue of crafting sound and complete proof calculi for partial and total correctness, let us briefly give examples of typical sorts of specifications which we would like to be able to prove.

Examples 4.7

1. Let `Succ` be the program

```
    a = x + 1;
    if (a - 1 == 0) {
        y = 1;
    } else {
        y = a;
    }
```

The program `Succ` satisfies the specification $\llparenthesis \top \rrparenthesis$ `Succ` $\llparenthesis y = (x + 1) \rrparenthesis$ under partial and total correctness, so if we think of x as input and y as output, then `Succ` computes the successor function. Note that this code is far from optimal.

In fact, it is a rather roundabout way of implementing the successor function. Despite this non-optimality, our proof rules need to be able to prove this program behaviour.

2. The program `Fac1` from Example 4.2 terminates only if x is initially non-negative – why? Let us look at what properties of `Fac1` we expect to be able to prove.

   We should be able to prove that $\vDash_{\mathsf{tot}} \llparenthesis x \geq 0 \rrparenthesis\ \texttt{Fac1}\ \llparenthesis y = x! \rrparenthesis$ holds. It states that, provided $x \geq 0$, `Fac1` terminates with the result $y = x!$. However, the stronger statement that $\vDash_{\mathsf{tot}} \llparenthesis \top \rrparenthesis\ \texttt{Fac1}\ \llparenthesis y = x! \rrparenthesis$ holds should not be provable, because `Fac1` does not terminate for negative values of x.

   For partial correctness, both statements $\vDash_{\mathsf{par}} \llparenthesis x \geq 0 \rrparenthesis\ \texttt{Fac1}\ \llparenthesis y = x! \rrparenthesis$ and $\vDash_{\mathsf{par}} \llparenthesis \top \rrparenthesis\ \texttt{Fac1}\ \llparenthesis y = x! \rrparenthesis$ should be provable since they hold.

Definition 4.8 1. If the partial correctness of triples $\llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis$ can be proved in the partial-correctness calculus we develop in this chapter, we say that the sequent $\vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis$ is valid.

2. Similarly, if it can be proved in the total-correctness calculus to be developed in this chapter, we say that the sequent $\vdash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis$ is valid.

Thus, $\vDash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis$ holds if P is partially correct, while the validity of $\vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis$ means that P can be proved to be partially-correct by our calculus. The first one means it is actually correct, while the second one means it is provably correct according to our calculus.

If our calculus is any good, then the relation $\vdash_{\mathsf{par}}$ should be contained in $\vDash_{\mathsf{par}}$! More precisely, we will say that our calculus is *sound* if, whenever it tells us something can be proved, that thing is indeed true. Thus, it is sound if it doesn't tell us that false things can be proved. Formally, we write that $\vdash_{\mathsf{par}}$ is sound if

$$\vDash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis \text{ holds whenever } \vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis \text{ is valid}$$

for all ϕ, ψ and P; and, similarly, $\vdash_{\mathsf{tot}}$ is sound if

$$\vDash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis \text{ holds whenever } \vdash_{\mathsf{tot}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis \text{ is valid}$$

for all ϕ, ψ and P. We say that a calculus is *complete* if it is able to prove everything that is true. Formally, $\vdash_{\mathsf{par}}$ is complete if

$$\vdash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis \text{ is valid whenever } \vDash_{\mathsf{par}} \llparenthesis \phi \rrparenthesis\ P\ \llparenthesis \psi \rrparenthesis \text{ holds}$$

for all ϕ, ψ and P; and similarly for $\vdash_{\mathsf{tot}}$ being complete.

In Chapters 1 and 2, we said that soundness is relatively easy to show, since typically the soundness of individual proof rules can be established independently of the others. Completeness, on the other hand, is harder to show since it depends on the entire set of proof rules cooperating together. The same situation holds for the program logic we introduce in this chapter. Establishing its soundness is simply a matter of considering each rule in turn – done in exercise 3 on page 303 – whereas establishing its (relative) completeness is harder and beyond the scope of this book.

4.2.4 Program variables and logical variables

The variables which we have seen so far in the programs that we verify are called *program variables*. They can also appear in the preconditions and postconditions of specifications. Sometimes, in order to formulate specifications, we need to use other variables which do not appear in programs.

Examples 4.9

1. Another version of the factorial program might have been Fac2:

   ```
   y = 1;
   while (x != 0) {
      y = y * x;
      x = x - 1;
      }
   ```

   Unlike the previous version, it 'consumes' the input x. Nevertheless, it correctly calculates the factorial of x and stores the value in y; and we would like to express that as a Hoare triple. However, it is not a good idea to write $\llparenthesis x \geq 0 \rrparenthesis$ Fac2 $\llparenthesis y = x! \rrparenthesis$ because, if the program terminates, then x will be 0 and y will be the factorial of the initial value of x.

   We need a way of remembering the initial value of x, to cope with the fact that it is modified by the program. Logical variables achieve just that: in the specification $\llparenthesis x = x_0 \wedge x \geq 0 \rrparenthesis$ Fac2 $\llparenthesis y = x_0! \rrparenthesis$ the x_0 is a logical variable and we read it as being universally quantified in the precondition. Therefore, this specification reads: for all integers x_0, if x equals x_0, $x \geq 0$ and we run the program such that it terminates, then the resulting state will satisfy y equals $x_0!$. This works since x_0 cannot be modified by Fac2 as x_0 does not occur in Fac2.
2. Consider the program Sum:

   ```
   z = 0;
   while (x > 0) {
       z = z + x;
       x = x - 1;
       }
   ```

   This program adds up the first x integers and stores the result in z. Thus, $\llparenthesis x = 3 \rrparenthesis$ Sum $\llparenthesis z = 6 \rrparenthesis$, $\llparenthesis x = 8 \rrparenthesis$ Sum $\llparenthesis z = 36 \rrparenthesis$ etc. We know from Theorem 1.31 on page 41 that $1 + 2 + \cdots + x = x(x + 1)/2$ for all $x \geq 0$, so we would like to express, as a Hoare triple, that the value of z upon termination is $x_0(x_0+1)/2$ where x_0 is the initial value of x. Thus, we write $(\!|x = x_0 \wedge x \geq 0|\!)$ `Sum` $(\!|z = x_0(x_0+1)/2|\!)$.

Variables like x_0 in these examples are called *logical variables*, because they occur only in the logical formulas that constitute the precondition and postcondition; they do not occur in the code to be verified. The state of the system gives a value to each program variable, but not for the logical variables. Logical variables take a similar role to the dummy variables of the rules for ∀i and ∃e in Chapter 2.

Definition 4.10 For a Hoare triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$, its set of logical variables are those variables that are free in ϕ or ψ; and don't occur in P.

4.3 Proof calculus for partial correctness

The proof calculus which we now present goes back to R. Floyd and C. A. R. Hoare. In the next subsection, we specify proof rules for each of the grammar clauses for commands. We could go on to use these proof rules directly, but it turns out to be more convenient to present them in a different form, suitable for the construction of proofs known as *proof tableaux*. This is what we do in the subsection following the next one.

4.3.1 Proof rules

The proof rules for our calculus are given in Figure 4.1. They should be interpreted as rules that allow us to pass from simple assertions of the form $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ to more complex ones. The rule for assignment is an axiom as it has no premises. This allows us to construct some triples out of nothing, to get the proof going. Complete proofs are trees, see page 274 for an example.

Composition. Given specifications for the program fragments C_1 and C_2, say

$$(\!|\phi|\!)\ C_1\ (\!|\eta|\!) \quad \text{and} \quad (\!|\eta|\!)\ C_2\ (\!|\psi|\!),$$

where the postcondition of C_1 is also the precondition of C_2, the proof rule for sequential composition shown in Figure 4.1 allows us to derive a specification for $C_1; C_2$, namely

$$(\!|\phi|\!)\ \ C_1; C_2\ \ (\!|\psi|\!).$$

$$\frac{\llparenthesis \phi \rrparenthesis\; C_1\; \llparenthesis \eta \rrparenthesis \qquad \llparenthesis \eta \rrparenthesis\; C_2\; \llparenthesis \psi \rrparenthesis}{\llparenthesis \phi \rrparenthesis\;\; C_1; C_2\;\; \llparenthesis \psi \rrparenthesis}\ \text{Composition}$$

$$\frac{}{\llparenthesis \psi[E/x] \rrparenthesis\; x = E\; \llparenthesis \psi \rrparenthesis}\ \text{Assignment}$$

$$\frac{\llparenthesis \phi \wedge B \rrparenthesis\; C_1\; \llparenthesis \psi \rrparenthesis \qquad \llparenthesis \phi \wedge \neg B \rrparenthesis\; C_2\; \llparenthesis \psi \rrparenthesis}{\llparenthesis \phi \rrparenthesis\; \texttt{if}\; B\; \{C_1\}\; \texttt{else}\; \{C_2\}\; \llparenthesis \psi \rrparenthesis}\ \text{If-statement}$$

$$\frac{\llparenthesis \psi \wedge B \rrparenthesis\; C\; \llparenthesis \psi \rrparenthesis}{\llparenthesis \psi \rrparenthesis\; \texttt{while}\; B\; \{C\}\; \llparenthesis \psi \wedge \neg B \rrparenthesis}\ \text{Partial-while}$$

$$\frac{\vdash_{\mathrm{AR}} \phi' \to \phi \qquad \llparenthesis \phi \rrparenthesis\; C\; \llparenthesis \psi \rrparenthesis \qquad \vdash_{\mathrm{AR}} \psi \to \psi'}{\llparenthesis \phi' \rrparenthesis\; C\; \llparenthesis \psi' \rrparenthesis}\ \text{Implied}$$

Figure 4.1. Proof rules for partial correctness of Hoare triples.

Thus, if we know that C_1 takes ϕ-states to η-states and C_2 takes η-states to ψ-states, then running C_1 and C_2 in that sequence will take ϕ-states to ψ-states.

Using the proof rules of Figure 4.1 in program verification, we have to read them bottom-up: e.g. in order to prove $\llparenthesis \phi \rrparenthesis\; C_1; C_2\; \llparenthesis \psi \rrparenthesis$, we need to find an appropriate η and prove $\llparenthesis \phi \rrparenthesis\; C_1\; \llparenthesis \eta \rrparenthesis$ and $\llparenthesis \eta \rrparenthesis\; C_2\; \llparenthesis \psi \rrparenthesis$. If $C_1; C_2$ runs on input satisfying ϕ and we need to show that the store satisfies ψ after its execution, then we hope to show this by splitting the problem into two. After the execution of C_1, we have a store satisfying η which, considered as input for C_2, should result in an output satisfying ψ. We call η a *midcondition.*

Assignment. The rule for assignment has no premises and is therefore an axiom of our logic. It tells us that, if we wish to show that ψ holds in the state after the assignment `x = ` E, we must show that $\psi[E/x]$ holds before the assignment; $\psi[E/x]$ denotes the formula obtained by taking ψ and replacing all free occurrences of x with E as defined on page 105. We read the stroke as 'in place of;' thus, $\psi[E/x]$ is ψ with E in place of x. Several explanations may be required to understand this rule.

- At first sight, it looks as if the rule has been stated in reverse; one might expect that, if ψ holds in a state in which we perform the assignment `x = ` E, then surely

$\psi[E/x]$ holds in the resulting state, i.e. we just replace x by E. This is wrong. It is true that the assignment `x = ` E replaces the value of x in the starting state by E, but that does not mean that we replace occurrences of x in a *condition on* the starting state by E.

For example, let ψ be $x = 6$ and E be 5. Then $(\!|\psi|\!)$ `x = 5` $(\!|\psi[x/E]|\!)$ does *not* hold: given a state in which x equals 6, the execution of `x = 5` results in a state in which x equals 5. But $\psi[x/E]$ is the formula $5 = 6$ which holds in no state.

The right way to understand the Assignment rule is to think about what you would have to prove about the initial state in order to prove that ψ holds in the resulting state. Since ψ will – in general – be saying something about the value of x, whatever it says about that value must have been true of E, since in the resulting state the value of x is E. Thus, ψ with E in place of x – which says whatever ψ says about x but applied to E – must be true in the initial state.

- The axiom $(\!|\psi[E/x]|\!)\; x = E\; (\!|\psi|\!)$ is best applied backwards than forwards in the verification process. That is to say, if we know ψ and we wish to find ϕ such that $(\!|\phi|\!)\; x = E\; (\!|\psi|\!)$, it is easy: we simply set ϕ to be $\psi[E/x]$; but, if we know ϕ and we want to find ψ such that $(\!|\phi|\!)\; x = E\; (\!|\psi|\!)$, there is no easy way of getting a suitable ψ. This backwards characteristic of the assignment and the composition rule will be important when we look at how to construct proofs; we will work from the end of a program to its beginning.
- If we apply this axiom in this backwards fashion, then it is completely mechanical to apply. It just involves doing a substitution. That means we could get a computer to do it for us. Unfortunately, that is not true for all the rules; application of the rule for while-statements, for example, requires ingenuity. Therefore a computer can at best assist us in performing a proof by carrying out the mechanical steps, such as application of the assignment axiom, while leaving the steps that involve ingenuity to the programmer.
- Observe that, in computing $\psi[E/x]$ from ψ, we replace all the free occurrences of x in ψ. Note that there cannot be problems caused by *bound* occurrences, as seen in Example 2.9 on page 106, *provided that preconditions and postconditions quantify over logical variables only.* For obvious reasons, this is recommended practice.

Examples 4.11

1. Suppose P is the program `x = 2`. The following are instances of axiom Assignment:

   a $(\!|2 = 2|\!) P (\!|x = 2|\!)$
   b $(\!|2 = 4|\!) P (\!|x = 4|\!)$
   c $(\!|2 = y|\!) P (\!|x = y|\!)$
   d $(\!|2 > 0|\!) P (\!|x > 0|\!)$.

These are all correct statements. Reading them backwards, we see that they say:

a If you want to prove $x = 2$ after the assignment `x = 2`, then we must be able to prove that $2 = 2$ before it. Of course, 2 is equal to 2, so proving it shouldn't present a problem.

b If you wanted to prove that $x = 4$ after the assignment, the only way in which it would work is if $2 = 4$; however, unfortunately it is not. More generally, $(\!|\bot|\!)\ x = E\ (\!|\psi|\!)$ holds for any E and ψ – why?

c If you want to prove $x = y$ after the assignment, you will need to prove that $2 = y$ before it.

d To prove $x > 0$, we'd better have $2 > 0$ prior to the execution of P.

2. Suppose P is $\mathtt{x} = \mathtt{x} + \mathtt{1}$. By choosing various postconditions, we obtain the following instances of the assignment axiom:

a $(\!|x + 1 = 2|\!)P(\!|x = 2|\!)$
b $(\!|x + 1 = y|\!)P(\!|x = y|\!)$
c $(\!|x + 1 + 5 = y|\!)P(\!|x + 5 = y|\!)$
d $(\!|x + 1 > 0 \wedge y > 0|\!)P(\!|x > 0 \wedge y > 0|\!)$.

Note that the precondition obtained by performing the substitution can often be simplified. The proof rule for implications below will allow such simplifications which are needed to make preconditions appreciable by human consumers.

If-statements. The proof rule for if-statements allows us to prove a triple of the form

$$(\!|\phi|\!)\ \mathtt{if}\ B\ \{C_1\}\ \mathtt{else}\ \{C_2\}\ (\!|\psi|\!)$$

by decomposing it into two triples, subgoals corresponding to the cases of B evaluating to true and to false. Typically, the precondition ϕ will not tell us anything about the value of the boolean expression B, so we have to consider both cases. If B is true in the state we start in, then C_1 is executed and hence C_1 will have to translate ϕ states to ψ states; alternatively, if B is false, then C_2 will be executed and will have to do that job. Thus, we have to prove that $(\!|\phi \wedge B|\!)\ C_1\ (\!|\psi|\!)$ and $(\!|\phi \wedge \neg B|\!)\ C_2\ (\!|\psi|\!)$. Note that the preconditions are augmented by the knowledge that B is true and false, respectively. This additional information is often crucial for completing the respective subproofs.

While-statements. The rule for while-statements given in Figure 4.1 is arguably the most complicated one. The reason is that the while-statement is the most complicated construct in our language. It is the only command that 'loops,' i.e. executes the same piece of code several times. Also, unlike as the for-statement in languages like Java we cannot generally predict how many times while-statements will ‘loop’ around, or even whether they will terminate at all.

The key ingredient in the proof rule for Partial-while is the ‘invariant’ ψ. In general, the body C of the command `while (`B`)` $\{C\}$ changes the values of the variables it manipulates; but the invariant expresses a relationship between those values which is preserved by any execution of C. In the proof rule, ψ expresses this invariant; the rule’s premise, $(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)$, states that, if ψ and B are true before we execute C, and C terminates, then ψ will be true after it. The conclusion of Partial-while states that, no matter how many times the body C is executed, if ψ is true initially and the while-statement terminates, then ψ will be true at the end. Moreover, since the while-statement has terminated, B will be false.

Implied. One final rule is required in our calculus: the rule Implied of Figure 4.1. It tells us that, if we have proved $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ and we have a formula ϕ' which implies ϕ and another one ψ' which is implied by ψ, then we should also be allowed to prove that $(\!|\phi'|\!)\ P\ (\!|\psi'|\!)$. A sequent $\vdash_{\mathrm{AR}} \phi \rightarrow \phi'$ is valid iff there is a proof of ϕ' in the natural deduction calculus for predicate logic, where ϕ and standard laws of arithmetic – e.g. $\forall x\,(x = x + 0)$ – are premises. Note that the rule Implied allows the precondition to be strengthened (thus, we *assume* more than we need to), while the postcondition is weakened (i.e. we *conclude* less than we are entitled to). If we tried to do it the other way around, weakening the precondition or strengthening the postcondition, then we would conclude things which are incorrect – see exercise 9(a) on page 300.

The rule Implied acts as a link between program logic and a suitable extension of predicate logic. It allows us to import proofs in predicate logic enlarged with the basic facts of arithmetic, which are required for reasoning about integer expressions, into the proofs in program logic.

4.3.2 Proof tableaux

The proof rules presented in Figure 4.1 are not in a form which is easy to use in examples. To illustrate this point, we present an example of a proof in Figure 4.2; it is a proof of the triple $(\!|\top|\!)$ `Fac1` $(\!|y = x!|\!)$ where `Fac1` is the factorial program given in Example 4.2. This proof abbreviates rule names; and drops the bars and names for Assignment as well as sequents for $\vdash_{\mathrm{AR}}$ in all applications of the Implied rule. We have not yet presented enough information for the reader to complete such a proof on her own, but she can at least use the proof rules in Figure 4.1 to check whether all rule instances of that proof are permissible, i.e. match the required pattern.

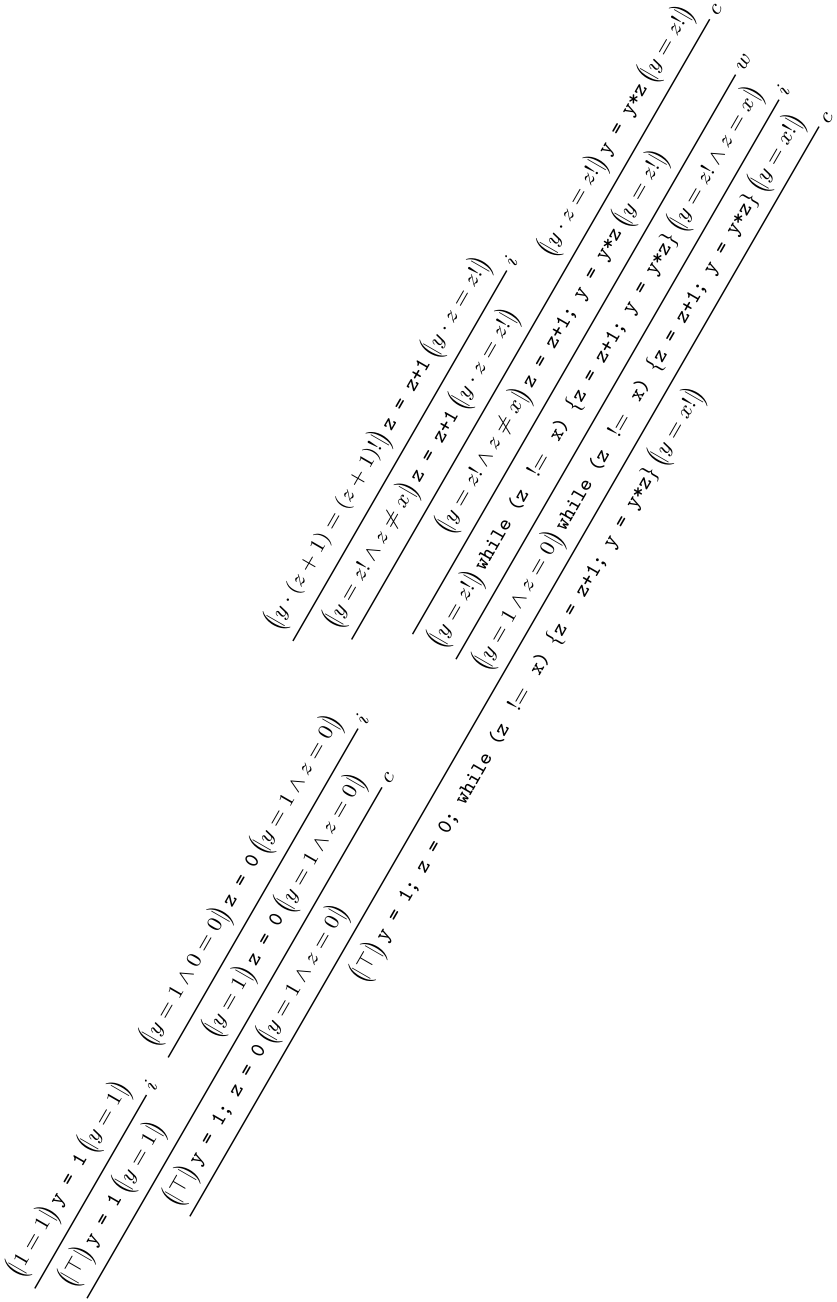

$(\!|1 = 1|\!)$ y = 1 $(\!|y = 1|\!)$
i
$(\!|\top|\!)$ y = 1 $(\!|y = 1|\!)$
$(\!|y = 1 \wedge 0 = 0|\!)$ z = 0 $(\!|y = 1 \wedge z = 0|\!)$
i
$(\!|y = 1|\!)$ z = 0 $(\!|y = 1 \wedge z = 0|\!)$
c
$(\!|\top|\!)$ y = 1; z = 0 $(\!|y = 1 \wedge z = 0|\!)$
$(\!|y \cdot (z+1) = (z+1)!|\!)$ z = z+1 $(\!|y \cdot z = z!|\!)$
i
$(\!|y = z! \wedge z \neq x|\!)$ z = z+1 $(\!|y \cdot z = z!|\!)$
$(\!|y \cdot z = z!|\!)$ y = y*z $(\!|y = z!|\!)$
c
$(\!|y = z! \wedge z \neq x|\!)$ z = z+1; y = y*z $(\!|y = z!|\!)$
w
$(\!|y = z!|\!)$ while (z != x) {z = z+1; y = y*z} $(\!|y = z! \wedge z = x|\!)$
i
$(\!|y = 1 \wedge z = 0|\!)$ while (z != x) {z = z+1; y = y*z} $(\!|y = x!|\!)$
c
$(\!|\top|\!)$ y = 1; z = 0; while (z != x) {z = z+1; y = y*z} $(\!|y = x!|\!)$

Figure 4.2. A partial-correctness proof for `Fac1` in tree form.

It should be clear that proofs in this form are unwieldy to work with. They will tend to be very wide and a lot of information is copied from one line to the next. Proving properties of programs which are longer than `Fac1` would be very difficult in this style. In Chapters 1, 2 and 5 we abandon representation of proofs as trees for similar reasons. The rule for sequential composition suggests a more convenient way of presenting proofs in program logic, called *proof tableaux*. We can think of any program of our core programming language as a sequence

$$\begin{array}{l} C_1; \\ C_2; \\ \cdot \\ \cdot \\ \cdot \\ C_n \end{array}$$

where none of the commands C_i is a composition of smaller programs, i.e. all of the C_i above are either assignments, if-statements or while-statements. Of course, we allow the if-statements and while-statements to have embedded compositions.

Let P stand for the program $C_1; C_2; \dots ; C_{n-1}; C_n$. Suppose that we want to show the validity of $\vdash_{\mathsf{par}} \llparenthesis \phi_0 \rrparenthesis \; P \; \llparenthesis \phi_n \rrparenthesis$ for a precondition ϕ_0 and a postcondition ϕ_n. Then, we may split this problem into smaller ones by trying to find formulas ϕ_j $(0 < j < n)$ and prove the validity of $\vdash_{\mathsf{par}} \llparenthesis \phi_i \rrparenthesis \; C_{i+1} \; \llparenthesis \phi_{i+1} \rrparenthesis$ for $i = 0, 1, \dots, n-1$. This suggests that we should design a proof calculus which presents a proof of $\vdash_{\mathsf{par}} \llparenthesis \phi_0 \rrparenthesis \; P \; \llparenthesis \psi_n \rrparenthesis$ by interleaving formulas with code as in

$$\begin{array}{lll}
 & \llparenthesis \phi_0 \rrparenthesis & \\
C_1\,; & & \\
 & \llparenthesis \phi_1 \rrparenthesis & \qquad \texttt{justification} \\
C_2\,; & & \\
\cdot & & \\
\cdot & & \\
\cdot & & \\
 & \llparenthesis \phi_{n-1} \rrparenthesis & \qquad \texttt{justification} \\
C_n\,; & & \\
 & \llparenthesis \phi_n \rrparenthesis & \qquad \texttt{justification}
\end{array}$$

Against each formula, we write a justification, whose nature will be clarified shortly. Proof tableaux thus consist of the program code interleaved with formulas, which we call *midconditions*, that should hold at the point they are written.

Each of the transitions

$$\begin{array}{l} \quad (\!|\phi_i|\!) \\ C_{i+1} \\ \quad (\!|\phi_{i+1}|\!) \end{array}$$

will appeal to one of the rules of Figure 4.1, depending on whether C_{i+1} is an assignment, an if-statement or a while-statement. Note that this notation for proofs makes the proof rule for composition in Figure 4.1 implicit.

How should the intermediate formulas ϕ_i be found? In principle, it seems as though one could start from ϕ_0 and, using C_1, obtain ϕ_1 and continue working downwards. However, because the assignment rule works backwards, it turns out that it is more convenient to start with ϕ_n and work upwards, using C_n to obtain ϕ_{n-1} etc.

Definition 4.12 The process of obtaining ϕ_i from C_{i+1} and ϕ_{i+1} is called computing the weakest precondition of C_{i+1}, given the postcondition ϕ_{i+1}. That is to say, we are looking for the logically weakest formula whose truth at the beginning of the execution of C_{i+1} is enough to guarantee ϕ_{i+1}[4].

The construction of a proof tableau for $(\!|\phi|\!)\ C_1; \dots; C_n\ (\!|\psi|\!)$ typically consists of starting with the postcondition ψ and pushing it upwards through C_n, then $C_{n-1}, \dots$, until a formula ϕ' emerges at the top. Ideally, the formula ϕ' represents the weakest precondition which guarantees that the ψ will hold if the composed program $C_1; C_2; \dots; C_{n-1}; C_n$ is executed and terminates. The weakest precondition ϕ' is then checked to see whether it follows from the given precondition ϕ. Thus, we appeal to the Implied rule of Figure 4.1.

Before a discussion of how to find invariants for while-statement, we now look at the assignment and the if-statement to see how the weakest precondition is calculated for each one.

Assignment. The assignment axiom is easily adapted to work for proof tableaux. We write it thus:

[4] ϕ is weaker than ψ means that ϕ is implied by ψ in predicate logic enlarged with the basic facts about arithmetic: the sequent $\vdash_{\mathrm{AR}} \psi \to \phi$ is valid. We want the weakest formula, because we want to impose as few constraints as possible on the preceding code. In some cases, especially those involving while-statements, it might not be possible to extract the logically weakest formula. We just need one which is sufficiently weak to allow us to complete the proof at hand.

$$\begin{array}{ll} \quad \llparenthesis \psi[E/x] \rrparenthesis & \\ \texttt{x = } E & \\ \quad \llparenthesis \psi \rrparenthesis & \qquad \textsf{Assignment} \end{array}$$

The justification is written against the ψ, since, once the proof has been constructed, we want to read it in a forwards direction. The construction itself proceeds in a backwards direction, because that is the way the assignment axiom facilitates.

Implied. In tableau form, the Implied rule allows us to write one formula ϕ_2 directly underneath another one ϕ_1 with no code in between, provided that ϕ_1 implies ϕ_2 in that the sequent $\vdash_{\mathrm{AR}} \phi_1 \to \phi_2$ is valid. Thus, the Implied rule acts as an interface between predicate logic with arithmetic and program logic. This is a surprising and crucial insight. Our proof calculus for partial correctness is a hybrid system which interfaces with another proof calculus via the Implied proof rule *only*.

When we appeal to the Implied rule, we will usually not explicitly write out the proof of the implication in predicate logic, for this chapter focuses on the program logic. Mostly, the implications we typically encounter will be easy to verify.

The Implied rule is often used to simplify formulas that are generated by applications of the other rules. It is also used when the weakest precondition ϕ' emerges by pushing the postcondition upwards through the whole program. We use the Implied rule to show that the given precondition implies the weakest precondition. Let's look at some examples of this.

Examples 4.13

1. We show that $\vdash_{\mathsf{par}} \llparenthesis y = 5 \rrparenthesis$ `x = y + 1` $\llparenthesis x = 6 \rrparenthesis$ is valid:

$$\begin{array}{ll} \quad \llparenthesis y = 5 \rrparenthesis & \\ \quad \llparenthesis y + 1 = 6 \rrparenthesis & \qquad \textsf{Implied} \\ \texttt{x = y + 1} & \\ \quad \llparenthesis x = 6 \rrparenthesis & \qquad \textsf{Assignment} \end{array}$$

The proof is constructed from the bottom upwards. We start with $\llparenthesis x = 6 \rrparenthesis$ and, using the assignment axiom, we push it upwards through `x = y + 1`. This means substituting $y + 1$ for all occurrences of x, resulting in $\llparenthesis y + 1 = 6 \rrparenthesis$. Now, we compare this with the given precondition $\llparenthesis y = 5 \rrparenthesis$. The given precondition and the arithmetic fact $5 + 1 = 6$ imply it, so we have finished the proof.

Although the proof is constructed bottom-up, its justifications make sense when read top-down: the second line is implied by the first and the fourth follows from the second by the intervening assignment.

2. We prove the validity of $\vdash_{\mathsf{par}} (\!|\, y < 3 \,|\!)$ `y = y + 1` $(\!|\, y < 4 \,|\!)$:

$$\begin{array}{lll} & (\!|\, y < 3 \,|\!) & \\ & (\!|\, y + 1 < 4 \,|\!) & \textsf{Implied} \\ \texttt{y = y + 1;} & & \\ & (\!|\, y < 4 \,|\!) & \textsf{Assignment} \end{array}$$

Notice that Implied always refers to the immediately preceding line. As already remarked, proofs in program logic generally combine two logical levels: the first level is directly concerned with proof rules for programming constructs such as the assignment statement; the second level is ordinary entailment familiar to us from Chapters 1 and 2 plus facts from arithmetic – here that $y < 3$ implies $y + 1 < 3 + 1 = 4$.

We may use ordinary logical and arithmetic implications to change a certain condition ϕ to any condition ϕ' which is implied by ϕ for reasons which have nothing to do with the given code. In the example above, ϕ was $y < 3$ and the implied formula ϕ' was then $y + 1 < 4$. The validity of $\vdash_{\mathrm{AR}} (y < 3) \to (y + 1 < 4)$ is rooted in general facts about integers and the relation $<$ defined on them. Completely formal proofs would require separate proofs attached to all instances of the rule Implied. As already said, we won't do that here as this chapter focuses on aspects of proofs which deal directly with code.

3. For the sequential composition of assignment statements

```
z = x;
z = z + y;
u = z;
```

our goal is to show that u stores the sum of x and y after this sequence of assignments terminates. Let us write P for the code above. Thus, we mean to prove $\vdash_{\mathsf{par}} (\!|\, \top \,|\!)\ P\ (\!|\, u = x + y \,|\!)$.

We construct the proof by starting with the postcondition $u = x + y$ and pushing it up through the assignments, in reverse order, using the assignment rule.

- Pushing it up through `u = z` involves replacing all occurrences of u by z, resulting in $z = x + y$. We thus have the proof fragment

$$\begin{array}{lll} & (\!|\, z = x + y \,|\!) & \\ \texttt{u = z;} & & \\ & (\!|\, u = x + y \,|\!) & \textsf{Assignment} \end{array}$$

- Pushing $z = x + y$ upwards through `z = z + y` involves replacing z by $z + y$, resulting in $z + y = x + y$.

- Pushing that upwards through `z = x` involves replacing z by x, resulting in $x+y=x+y$. The proof fragment now looks like this:

$\llparenthesis x+y=x+y \rrparenthesis$

`z = x;`

$\llparenthesis z+y=x+y \rrparenthesis$ Assignment

`z = z + y;`

$\llparenthesis z=x+y \rrparenthesis$ Assignment

`u = z;`

$\llparenthesis u=x+y \rrparenthesis$ Assignment

The weakest precondition that thus emerges is $x+y=x+y$; we have to check that this follows from the given precondition $\top$. This means checking that any state that satisfies $\top$ also satisfies $x+y=x+y$. Well, $\top$ is satisfied in all states, but so is $x+y=x+y$, so the sequent $\vdash_{\mathrm{AR}} \top \rightarrow (x+y=x+y)$ is valid. The final completed proof therefore looks like this:

$\llparenthesis \top \rrparenthesis$

$\llparenthesis x+y=x+y \rrparenthesis$ Implied

`z = x;`

$\llparenthesis z+y=x+y \rrparenthesis$ Assignment

`z = z + y;`

$\llparenthesis z=x+y \rrparenthesis$ Assignment

`u = z;`

$\llparenthesis u=x+y \rrparenthesis$ Assignment

and we can now read it from the top down.

The application of the axiom Assignment requires some care. We describe two pitfalls which the unwary may fall into, if the rule is not applied correctly.

- Consider the example 'proof'

$\llparenthesis x+1=x+1 \rrparenthesis$

`x = x + 1;`

$\llparenthesis x=x+1 \rrparenthesis$ Assignment

which uses the rule for assignment incorrectly. Pattern matching with the assignment axiom means that ψ has to be $x=x+1$, the expression E is $x+1$ and $\psi[E/x]$ is $x+1=x+1$. However, $\psi[E/x]$ is obtained by replacing *all* occurrences of x in ψ by E, thus, $\psi[E/x]$ would have to be equal to $x+1=x+1+1$. Therefore, the corrected proof

$(\!|x+1=x+1+1|\!)$

`x = x + 1;`

$(\!|x=x+1|\!)$ Assignment

shows that $\vdash_{\mathsf{par}} (\!|x+1=x+1+1|\!)$ `x = x + 1` $(\!|x=x+1|\!)$ is valid.
As an aside, this corrected proof is not very useful. The triple says that, if $x+1=(x+1)+1$ holds in a state and the assignment `x = x + 1` is executed and terminates, then the resulting state satisfies $x=x+1$; but, since the precondition $x+1=x+1+1$ can never be true, this triple tells us nothing informative about the assignment.

- Another way of using the proof rule for assignment incorrectly is by allowing additional assignments to happen in between $\psi[E/x]$ and `x =` E, as in the 'proof'

  $(\!|x+2=y+1|\!)$

  `y = y + 1000001;`

  `x = x + 2;`

  $(\!|x=y+1|\!)$ Assignment

  This is not a correct application of the assignment rule, since an additional assignment happens in line 2 right before the actual assignment to which the inference in line 4 applies. This additional assignment makes this reasoning unsound: line 2 overwrites the current value in y to which the equation in line 1 is referring. Clearly, $x+2=y+1$ won't be true any longer. Therefore, we are allowed to use the proof rule for assignment only if there is no additional code between the precondition $\psi[E/x]$ and the assignment `x =` E.

If-statements. We now consider how to push a postcondition upwards through an if-statement. Suppose we are given a condition ψ and a program fragment `if (`B`)` $\{C_1\}$ `else` $\{C_2\}$. We wish to calculate the weakest ϕ such that

$$(\!|\phi|\!)\ \texttt{if (}B\texttt{)}\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!).$$

This ϕ may be calculated as follows.

1. Push ψ upwards through C_1; let's call the result ϕ_1. (Note that, since C_1 may be a sequence of other commands, this will involve appealing to other rules. If C_1 contains another if-statement, then this step will involve a 'recursive call' to the rule for if-statements.)
2. Similarly, push ψ upwards through C_2; call the result ϕ_2.
3. Set ϕ to be $(B \to \phi_1) \land (\neg B \to \phi_2)$.

Example 4.14 Let us see this proof rule at work on the non-optimal code for `Succ` given earlier in the chapter. Here is the code again:

```
a = x + 1;
if (a - 1 == 0)  {
  y = 1;
} else {
  y = a;
}
```

We want to show that $\vdash_{\mathsf{par}} (\!|\top|\!)\ \texttt{Succ}\ (\!|y = x+1|\!)$ is valid. Note that this program is the sequential composition of an assignment and an if-statement. Thus, we need to obtain a suitable midcondition to put between the if-statement and the assignment.

We push the postcondition $y = x + 1$ upwards through the two branches of the if-statement, obtaining

- ϕ_1 is $1 = x + 1$;
- ϕ_2 is $a = x + 1$;

and obtain the midcondition $(a - 1 = 0 \to 1 = x + 1) \wedge (\neg(a - 1 = 0) \to a = x + 1)$ by appealing to a slightly different version of the rule If-statement:

$$\frac{(\!|\phi_1|\!)\ C_1\ (\!|\psi|\!) \qquad (\!|\phi_2|\!)\ C_2\ (\!|\psi|\!)}{(\!|(B \to \phi_1) \wedge (\neg B \to \phi_2)|\!)\ \texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!)}\ \text{If-Statement} \tag{4.7}$$

However, this rule can be derived using the proof rules discussed so far; see exercise 9(c) on page 301. The partial proof now looks like this:

```
  $(\!|\top|\!)$
  $(\!|?|\!)$                                                  ?
a = x + 1;
  $(\!|(a-1=0 \to 1=x+1) \wedge (\neg(a-1=0) \to a=x+1)|\!)$   ?
if (a - 1 == 0) {
      $(\!|1 = x+1|\!)$                                        If-Statement
    y = 1;
      $(\!|y = x+1|\!)$                                        Assignment
} else {
      $(\!|a = x+1|\!)$                                        If-Statement
    y = a;
      $(\!|y = x+1|\!)$                                        Assignment
}
  $(\!|y = x+1|\!)$                                            If-Statement
```

Continuing this example, we push the long formula above the if-statement through the assignment, to obtain

$$(x + 1 - 1 = 0 \to 1 = x + 1) \wedge (\neg(x + 1 - 1 = 0) \to x + 1 = x + 1) \tag{4.8}$$

We need to show that this is implied by the given precondition $\top$, i.e. that it is true in any state. Indeed, simplifying (4.8) gives

$$(x = 0 \to 1 = x + 1) \land (\neg(x = 0) \to x + 1 = x + 1)$$

and both these conjuncts, and therefore their conjunction, are clearly valid implications. The above proof now is completed as:

```
  $\llparenthesis \top \rrparenthesis$
  $\llparenthesis (x + 1 - 1 = 0 \to 1 = x + 1) \land (\neg(x + 1 - 1 = 0) \to x + 1 = x + 1) \rrparenthesis$    Implied
a = x + 1;
  $\llparenthesis (a - 1 = 0 \to 1 = x + 1) \land (\neg(a - 1 = 0) \to a = x + 1) \rrparenthesis$                Assignment
if (a - 1 == 0) {
      $\llparenthesis 1 = x + 1 \rrparenthesis$                                                                   If-Statement
    y = 1;
      $\llparenthesis y = x + 1 \rrparenthesis$                                                                   Assignment
} else {
      $\llparenthesis a = x + 1 \rrparenthesis$                                                                   If-Statement
    y = a;
      $\llparenthesis y = x + 1 \rrparenthesis$                                                                   Assignment
}
  $\llparenthesis y = x + 1 \rrparenthesis$                                                                       If-Statement
```

While-statements. Recall that the proof rule for partial correctness of while-statements was presented in the following form in Figure 4.1 – here we have written η instead of ψ:

$$\frac{\llparenthesis \eta \land B \rrparenthesis \; C \; \llparenthesis \eta \rrparenthesis}{\llparenthesis \eta \rrparenthesis \; \texttt{while } B \; \{C\} \; \llparenthesis \eta \land \neg B \rrparenthesis} \text{ Partial-while.} \tag{4.9}$$

Before we look at how Partial-while will be represented in proof tableaux, let us look in more detail at the ideas behind this proof rule. The formula η is chosen to be an invariant of the body C of the while-statement: provided the boolean guard B is true, if η is true before we start C, and C terminates, then it is also true at the end. This is what the premise $\llparenthesis \eta \land B \rrparenthesis \; C \; \llparenthesis \eta \rrparenthesis$ expresses.

Now suppose the while-statement executes a terminating run from a state that satisfies η; and that the premise of (4.9) holds.

- If B is false as soon as we embark on the while-statement, then we do not execute C at all. Nothing has happened to change the truth value of η, so we end the while-statement with $\eta \land \neg B$.

- If B is true when we embark on the while-statement, we execute C. By the premise of the rule in (4.9), we know η is true at the end of C.
  - if B is now false, we stop with $\eta \wedge \neg B$.
  - if B is true, we execute C again; η is again re-established. No matter how many times we execute C in this way, η is re-established at the end of each execution of C. The while-statement terminates if, and only if, B is false after some finite (zero including) number of executions of C, in which case we have $\eta \wedge \neg B$.

This argument shows that Partial-while is sound with respect to the satisfaction relation for partial correctness, in the sense that anything we prove using it is indeed true. However, as it stands it allows us to prove only things of the form $(\!|\eta|\!)\ \texttt{while}\ \texttt{(}B\texttt{)}\ \{C\}\ (\!|\eta \wedge \neg B|\!)$, i.e. triples in which the postcondition is the same as the precondition conjoined with $\neg B$. Suppose that we are required to prove

$$(\!|\phi|\!)\ \texttt{while}\ \texttt{(}B\texttt{)}\ \{\texttt{C}\}\ (\!|\psi|\!) \tag{4.10}$$

for some ϕ and ψ which are not related in that way. How can we use Partial-while in a situation like this?

The answer is that we must *discover* a suitable η, such that

1. $\vdash_{\mathrm{AR}} \phi \to \eta$,
2. $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$ and
3. $\vdash_{\mathsf{par}} (\!|\eta|\!)\ \texttt{while}\ \texttt{(}B\texttt{)}\ \{C\}\ (\!|\eta \wedge \neg B|\!)$

are all valid, where the latter is shown by means of Partial-while. Then, Implied infers that (4.10) is a valid partial-correctness triple.

The crucial thing, then, is the discovery of a suitable invariant η. It is a necessary step in order to use the proof rule Partial-while and in general it requires intelligence and ingenuity. This contrasts markedly with the case of the proof rules for if-statements and assignments, which are purely mechanical in nature: their usage is just a matter of symbol-pushing and does not require any deeper insight.

Discovery of a suitable invariant requires careful thought about what the while-statement is really doing. Indeed the eminent computer scientist, the late E. Dijkstra, said that to understand a while-statement is tantamount to knowing what its invariant is with respect to given preconditions and postconditions for that while-statement.

This is because a suitable invariant can be interpreted as saying that the intended computation performed by the while-statement is correct up to the current step of the execution. It then follows that, when the execution terminates, the entire computation is correct. Let us formalize invariants and then study how to discover them.

Definition 4.15 An invariant of the while-statement `while (`B`)` $\{C\}$ is a formula η such that $\vDash_{\mathsf{par}} (\!|\, \eta \wedge B \,|\!)\ C\ (\!|\, \eta \,|\!)$ holds; i.e. for all states l, if η and B are true in l and C is executed from state l and terminates, then η is again true in the resulting state.

Note that η does not have to be true continuously during the execution of C; in general, it will not be. All we require is that, if it is true before C is executed, then it is true (if and) when C terminates.

For any given while-statement there are several invariants. For example, $\top$ is an invariant for *any* while-statement; so is $\bot$, since the premise of the implication 'if $\bot \wedge B$ is true, then ... ' is false, so that implication is true. The formula $\neg B$ is also an invariant of `while (`B`) do` $\{C\}$; but most of these invariants are useless to us, because we are looking for an invariant η for which the sequents $\vdash_{\mathrm{AR}} \phi \to \eta$ and $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$, are valid, where ϕ and ψ are the preconditions and postconditions of the while-statement. Usually, this will single out just one of all the possible invariants – up to logical equivalence.

A useful invariant expresses a relationship between the variables manipulated by the body of the while-statement which is preserved by the execution of the body, even though the values of the variables themselves may change. The invariant can often be found by constructing a trace of the while-statement in action.

Example 4.16 Consider the program `Fac1` from page 262, annotated with location labels for our discussion:

```
      y = 1;
      z = 0;
  l1:   while (z != x) {
          z = z + 1;
          y = y * z;
  l2:   }
```

Suppose program execution begins in a store in which x equals 6. When the program flow first encounters the while-statement at location `l1`, z equals 0 and y equals 1, so the condition $z \neq x$ is true and the body is executed. Thereafter at location `l2`, z equals 1 and y equals 1 and the boolean guard is still true, so the body is executed again. Continuing in this way, we obtain the following trace:

after iteration	z at `l1`	y at `l1`	B at `l1`
0	0	1	true
1	1	1	true
2	2	2	true
3	3	6	true
4	4	24	true
5	5	120	true
6	6	720	false

The program execution stops when the boolean guard becomes false.

The invariant of this example is easy to see: it is '$y = z!$'. Every time we complete an execution of the body of the while-statement, this fact is true, even though the values of y and z have been changed. Moreover, this invariant has the needed properties. It is

- weak enough to be implied by the precondition of the while-statement, which we will shortly discover to be $y = 1 \wedge z = 0$ based on the initial assignments and their precondition $0! \stackrel{\mathrm{def}}{=} 1$,
- but also strong enough that, together with the negation of the boolean guard, it implies the postcondition '$y = x!$'.

That is to say, the sequents

$$\vdash_{\mathrm{AR}} (y = 1 \wedge z = 0) \to (y = z!) \text{ and } \vdash_{\mathrm{AR}} (y = z! \wedge x = z) \to (y = x!) \tag{4.11}$$

are valid.

As in this example, a suitable invariant is often discovered by looking at the logical structure of the postcondition. A complete proof of the factorial example in tree form, using this invariant, was given in Figure 4.2.

How should we use the while-rule in proof tableaux? We need to think about how to push an arbitrary postcondition ψ upwards through a while-statement to meet the precondition ϕ. The steps are:

1. Guess a formula η which you hope is a suitable invariant.
2. Try to prove that $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$ and $\vdash_{\mathrm{AR}} \phi \to \eta$ are valid, where B is the boolean guard of the while-statement. If both proofs succeed, go to 3. Otherwise (if at least one proof fails), go back to 1.
3. Push η upwards through the body C of the while-statement; this involves applying other rules dictated by the form of C. Let us name the formula that emerges η'.

4. Try to prove that $\vdash_{\mathrm{AR}} \eta \wedge B \to \eta'$ is valid; this proves that η is indeed an invariant. If you succeed, go to 5. Otherwise, go back to 1.
5. Now write η above the while-statement and write ϕ above that η, annotating that η with an instance of Implied based on the successful proof of the validity of $\vdash_{\mathrm{AR}} \phi \to \eta$ in 2. Mission accomplished!

Example 4.17 We continue the example of the factorial. The partial proof obtained by pushing $y = x!$ upwards through the while-statement – thus checking the hypothesis that $y = z!$ is an invariant – is as follows:

```
y = 1;
z = 0;
   $\llparenthesis y = z! \rrparenthesis$                       ?
while (z != x) {
         $\llparenthesis y = z! \wedge z \neq x \rrparenthesis$            Invariant Hyp. $\wedge$ guard
         $\llparenthesis y \cdot (z+1) = (z+1)! \rrparenthesis$    Implied
     z = z + 1;
         $\llparenthesis y \cdot z = z! \rrparenthesis$                    Assignment
     y = y * z;
         $\llparenthesis y = z! \rrparenthesis$                         Assignment
}
   $\llparenthesis y = x! \rrparenthesis$                       ?
```

Whether $y = z!$ is a suitable invariant depends on three things:

- The ability to prove that it is indeed an invariant, i.e. that $y = z!$ implies $y \cdot (z + 1) = (z + 1)!$. This is the case, since we just multiply each side of $y = z!$ by $z + 1$ and appeal to the inductive definition of $(z + 1)!$ in Example 4.2.
- The ability to prove that η is strong enough that it and the negation of the boolean guard together imply the postcondition; this is also the case, for $y = z!$ and $x = z$ imply $y = x!$.
- The ability to prove that η is weak enough to be established by the code leading up to the while-statement. This is what we prove by continuing to push the result upwards through the code preceding the while-statement.

Continuing, then: pushing $y = z!$ through `z = 0` results in $y = 0!$ and pushing that through `y = 1` renders $1 = 0!$. The latter holds in all states as $0!$ is defined to be 1, so it is implied by $\top$; our completed proof is:

```
    $\llparenthesis \top \rrparenthesis$
    $\llparenthesis 1 = 0! \rrparenthesis$                        Implied
y = 1;
    $\llparenthesis y = 0! \rrparenthesis$                        Assignment
z = 0;
    $\llparenthesis y = z! \rrparenthesis$                        Assignment
while (z != x) {
         $\llparenthesis y = z! \wedge z \neq x \rrparenthesis$             Invariant Hyp. $\wedge$ guard
         $\llparenthesis y \cdot (z+1) = (z+1)! \rrparenthesis$         Implied
     z = z + 1;
         $\llparenthesis y \cdot z = z! \rrparenthesis$                 Assignment
     y = y * z;
         $\llparenthesis y = z! \rrparenthesis$                     Assignment
}
    $\llparenthesis y = z! \wedge \neg(z \neq x) \rrparenthesis$               Partial-while
    $\llparenthesis y = x! \rrparenthesis$                        Implied
```

4.3.3 A case study: minimal-sum section

We practice the proof rule for while-statements once again by verifying a program which computes the minimal-sum section of an array of integers. For that, let us extend our core programming language with arrays of integers[5]. For example, we may declare an array

```
int a[n];
```

whose name is `a` and whose fields are accessed by `a[0]`, `a[1]`, ..., `a[n-1]`, where `n` is some constant. Generally, we allow any integer expression E to compute the field index, as in `a[E]`. It is the programmer's responsibility to make sure that the value computed by `E` is always within the array bounds.

Definition 4.18 Let $a[0], \ldots, a[n-1]$ be the integer values of an array `a`. A section of `a` is a continuous piece $a[i], \ldots, a[j]$, where $0 \leq i \leq j < n$. We

[5] We only read from arrays in the program `Min_Sum` which follows. Writing to arrays introduces additional problems because an array element can have several syntactically different names and this has to be taken into account by the calculus.

write $S_{i,j}$ for the sum of that section: $a[i] + a[i+1] + \cdots + a[j]$. A minimal-sum section is a section $a[i], \dots, a[j]$ of a such that the sum $S_{i,j}$ is less than or equal to the sum $S_{i',j'}$ of any other section $a[i'], \dots, a[j']$ of a.

Example 4.19 Let us illustrate these concepts on the example integer array $[-1, 3, 15, -6, 4, -5]$. Both $[3, 15, -6]$ and $[-6]$ are sections, but $[3, -6, 4]$ isn't since 15 is missing. A minimal-sum section for this particular array is $[-6, 4, -5]$ with sum -7; it is the only minimal-sum section in this case.

In general, minimal-sum sections need not be unique. For example, the array $[1, -1, 3, -1, 1]$ has two minimal-sum sections $[1, -1]$ and $[-1, 1]$ with minimal sum 0.

The task at hand is to

- write a program `Min_Sum`, written in our core programming language extended with integer arrays, which computes the sum of a minimal-sum section of a given array;
- make the informal requirement of this problem, given in the previous item, into a formal specification about the behaviour of `Min_Sum`;
- use our proof calculus for partial correctness to show that `Min_Sum` satisfies those formal specifications provided that it terminates.

There is an obvious program to do the job: we could list all the possible sections of a given array, then traverse that list to compute the sum of each section and keep the recent minimal sum in a storage location. For the example array $[-1, 3, -2]$, this results in the list

$$[-1],\ [-1,3],\ [-1,3,-2],\ [3],\ [3,-2],\ [-2]$$

and we see that only the last section $[-2]$ produces the minimal sum -2. This idea can easily be coded in our core programming language, but it has a serious drawback: the number of sections of a given array of size n is proportional to the square of n; if we also have to sum all those, then our task has worst-case time complexity of the order $n \cdot n^2 = n^3$. Computationally, this is an expensive price to pay, so we should inspect the problem more closely in order to see whether we can do better.

Can we compute the minimal sum over all sections in time proportional to n, by passing through the array just once? Intuitively, this seems difficult, since if we store just the minimal sum seen so far as we pass through the array, we may miss the opportunity of some large negative numbers later on because of some large positive numbers we encounter en route. For example, suppose the array is

$$[-8, 3, -65, 20, 45, -100, -8, 17, -4, -14].$$

Should we settle for $-8 + 3 - 65$, or should we try to take advantage of the -100 – remembering that we can pass through the array only once? In this case, the whole array is a section that gives us the smallest sum, but it is difficult to see how a program which passes through the array just once could detect this.

The solution is to store two values during the pass: the minimal sum seen so far (s in the program below) and also the minimal sum seen so far of *all* sections which end at the current point in the array (t below). Here is a program that is intended to do this:

```
k = 1;
t = a[0];
s = a[0];
while (k != n) {
   t = min(t + a[k], a[k]);
   s = min(s,t);
   k = k + 1;
}
```

where `min` is a function which computes the minimum of its two arguments as specified in exercise 10 on page 301. The variable k proceeds through the range of indexes of the array and t stores the minimal sum of sections that end at $a[k]$ – whenever the control flow of the program is about to evaluate the boolean expression of its while-statement. As each new value is examined, we can either add it to the current minimal sum, or decide that a lower minimal sum can be obtained by starting a new section. The variable s stores the minimal sum seen so far; it is computed as the minimum we have seen so far in the last step, or the minimal sum of sections that end at the current point.

As you can see, it not intuitively clear that this program is correct, warranting the use of our partial-correctness calculus to prove its correctness. Testing the program with a few examples is not sufficient to find all mistakes, however, and the reader would rightly not be convinced that this program really does compute the minimal-sum section in all cases. So let us try to use the partial-correctness calculus introduced in this chapter to prove it.

We formalise our requirement of the program as two specifications[6], written as Hoare triples.

S1. $(\!|\top|\!)$ `Min_Sum` $(\!|\forall i, j\,(0 \le i \le j < n \to s \le S_{i,j})|\!)$.
It says that, after the program terminates, s is less than or equal to, the sum of any section of the array. Note that i and j are logical variables in that they don't occur as program variables.

S2. $(\!|\top|\!)$ `Min_Sum` $(\!|\exists i, j\,(0 \le i \le j < n \land s = S_{i,j})|\!)$,
which says that there is a section whose sum is s.

If there is a section whose sum is s and no section has a sum less than s, then s is the sum of a minimal-sum section: the 'conjunction' of **S1** and **S2** give us the property we want.

Let us first prove **S1**. This begins with seeking a suitable invariant. As always, the following characteristics of invariants are a useful guide:

- Invariants express the fact that the computation performed so far by the while-statement is correct.
- Invariants typically have the same form as the desired postcondition of the while-statement.
- Invariants express relationships between the variables manipulated by the while-statement which are re-established each time the body of the while-statement is executed.

A suitable invariant in this case appears to be

$$\texttt{Inv1}(s, k) \overset{\text{def}}{=} \forall i, j\,(0 \le i \le j < k \to s \le S_{i,j}) \tag{4.12}$$

since it says that s is less than, or equal to, the minimal sum observed up to the current stage of the computation, represented by k. Note that it has the same form as the desired postcondition: we replaced the n by k, since the final value of k is n. Notice that i and j are quantified in the formula, because they are logical variables; k is a program variable. This justifies the notation `Inv1`(s, k) which highlights that the formula has only the program variables s and k as free variables and is similar to the use of `fun`-statements in Alloy in Chapter 2.

If we start work on producing a proof tableau with this invariant, we will soon find that it is not strong enough to do the job. Intuitively, this is because it ignores the value of t, which stores the minimal sum of all sections ending just before $a[k]$, which is crucial in the idea behind the program. A suitable invariant expressing that t is correct up to the current point of the

[6] The notation $\forall i, j$ abbreviates $\forall i \forall j$, and similarly for $\exists i, j$.

```
  $(\!|\top|\!)$
  $(\!|\mathtt{Inv1}(a[0],1) \wedge \mathtt{Inv2}(a[0],1)|\!)$                 Implied
k = 1;
  $(\!|\mathtt{Inv1}(a[0],k) \wedge \mathtt{Inv2}(a[0],k)|\!)$                 Assignment
t = a[0];
  $(\!|\mathtt{Inv1}(a[0],k) \wedge \mathtt{Inv2}(t,k)|\!)$                    Assignment
s = a[0];
  $(\!|\mathtt{Inv1}(s,k) \wedge \mathtt{Inv2}(t,k)|\!)$                       Assignment
while (k != n) {
      $(\!|\mathtt{Inv1}(s,k) \wedge \mathtt{Inv2}(t,k) \wedge k \neq n|\!)$            Invariant Hyp. ∧ guard
      $(\!|\mathtt{Inv1}(\min(s,\min(t+a[k],a[k])),k+1)$
         $\wedge\mathtt{Inv2}(\min(t+a[k],a[k]),k+1)|\!)$              Implied (Lemma 4.20)
    t = min(t + a[k], a[k]);
      $(\!|\mathtt{Inv1}(\min(s,t),k+1) \wedge \mathtt{Inv2}(t,k+1)|\!)$       Assignment
    s = min(s,t);
      $(\!|\mathtt{Inv1}(s,k+1) \wedge \mathtt{Inv2}(t,k+1)|\!)$               Assignment
    k = k + 1;
      $(\!|\mathtt{Inv1}(s,k) \wedge \mathtt{Inv2}(t,k)|\!)$                   Assignment
}
  $(\!|\mathtt{Inv1}(s,k) \wedge \mathtt{Inv2}(t,k) \wedge \neg\neg(k=n)|\!)$       Partial-while
  $(\!|\mathtt{Inv1}(s,n)|\!)$                                      Implied
```

Figure 4.3. Tableau proof for specification **S1** of Min_Sum.

computation is

$$\mathtt{Inv2}(t,k) \stackrel{\text{def}}{=} \forall i\,(0 \le i < k \to t \le S_{i,k-1}) \tag{4.13}$$

saying that t is not greater than the sum of any section ending in $a[k-1]$. Our invariant is the conjunction of these formulas, namely

$$\mathtt{Inv1}(s,k) \wedge \mathtt{Inv2}(t,k). \tag{4.14}$$

The completed proof tableau of **S1** for Min_Sum is given in Figure 4.3. The tableau is constructed by

- Proving that the candidate invariant (4.14) is indeed an invariant. This involves pushing it upwards through the body of the while-statement and showing that what emerges follows from the invariant and the boolean guard. This non-trivial implication is shown in the proof of Lemma 4.20.
- Proving that the invariant, together with the negation of the boolean guard, is strong enough to prove the desired postcondition. This is the last implication of the proof tableau.

- Proving that the invariant is established by the code before the while-statement. We simply push it upwards through the three initial assignments and check that the resulting formula is implied by the precondition of the specification, here $\top$.

As so often the case, in constructing the tableau, we find that two formulas meet; and we have to prove that the first one implies the second one. Sometimes this is easy and we can just note the implication in the tableau. For example, we readily see that $\top$ implies $\mathtt{Inv1}(a[0],1) \wedge \mathtt{Inv2}(a[0],1)$: k being 1 forces i and j to be zero in order that the assumptions in $\mathtt{Inv1}(a[0],k)$ and $\mathtt{Inv2}(a[0],k)$ be true. But this means that their conclusions are true as well. However, the proof obligation that the invariant hypothesis imply the precondition computed within the body of the while-statement reveals the complexity and ingenuity of this program and its justification needs to be taken off-line:

Lemma 4.20 Let s and t be any integers, n the length of the array `a`, and k an index of that array in the range of $0 < k < n$. Then $\mathtt{Inv1}(s,k) \wedge \mathtt{Inv2}(t,k) \wedge k \neq n$ implies

1. $\mathtt{Inv1}(\min(s, \min(t + a[k], a[k])), k+1)$ as well as
2. $\mathtt{Inv2}(\min(t + a[k], a[k]), k+1)$.

PROOF:

1. Take any i with $0 \leq i < k+1$; we will prove that $\min(t + a[k], a[k]) \leq S_{i,k}$. If $i < k$, then $S_{i,k} = S_{i,k-1} + a[k]$, so what we have to prove is $\min(t + a[k], a[k]) \leq S_{i,k-1} + a[k]$; but we know $t \leq S_{i,k-1}$, so the result follows by adding $a[k]$ to each side. Otherwise, $i = k$, $S_{i,k} = a[k]$ and the result follows.
2. Take any i and j with $0 \leq i \leq j < k+1$; we prove that $\min(s, t + a[k], a[k]) \leq S_{i,j}$. If $i \leq j < k$, then the result is immediate. Otherwise, $i \leq j = k$ and the result follows from part 1 of the lemma.

□

4.4 Proof calculus for total correctness

In the preceding section, we developed a calculus for proving *partial* correctness of triples $(\!|\phi|\!)\ P\ (\!|\psi|\!)$. In that setting, proofs come with a disclaimer: *only if* the program P terminates an execution does a proof of $\vdash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ tell us anything about that execution. Partial correctness does not tell us anything if P 'loops' indefinitely. In this section, we extend our proof calculus for partial correctness so that it also proves that programs terminate. In the previous section, we already pointed out that only the syntactic construct `while` B $\{C\}$ could be responsible for non-termination.

> *Therefore, the proof calculus for total correctness is the same as for partial correctness for all the rules except the rule for while-statements.*

A proof of total correctness for a while-statement will consist of two parts: the proof of partial correctness and a proof that the given while-statement terminates. Usually, it is a good idea to prove partial correctness first since this often provides helpful insights for a termination proof. However, some programs require termination proofs as premises for establishing *partial* correctness, as can be seen in exercise 1(d) on page 303.

The proof of termination usually has the following form. We identify an integer expression whose value can be shown to *decrease* every time we execute the body of the while-statement in question, but which is always non-negative. If we can find an expression with these properties, it follows that the while-statement must terminate; because the expression can only be decremented a finite number of times before it becomes 0. That is because there is only a finite number of integer values between 0 and the initial value of the expression.

Such integer expressions are called *variants*. As an example, for the program `Fac1` of Example 4.2, a suitable variant is $x - z$. The value of this expression is decremented every time the body of the while-statement is executed. When it is 0, the while-statement terminates.

We can codify this intuition in the following rule for total correctness which replaces the rule for the while statement:

$$\frac{(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\; C\; (\!|\eta \wedge 0 \leq E < E_0|\!)}{(\!|\eta \wedge 0 \leq E|\!)\; \texttt{while}\; B\; \{C\}\; (\!|\eta \wedge \neg B|\!)}\ \textsf{Total-while.} \tag{4.15}$$

In this rule, E is the expression whose value decreases with each execution of the body C. This is coded by saying that, if its value equals that of the logical variable E_0 before the execution of C, then it is strictly less than E_0 after it – yet still it remains non-negative. As before, η is the invariant.

We use the rule Total-while in tableaux similarly to how we use Partial-while, but note that the body of the rule C must now be shown to satisfy

$$(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\; C\; (\!|\eta \wedge 0 \leq E < E_0|\!).$$

When we push $\eta \wedge 0 \leq E < E_0$ upwards through the body, we have to prove that what emerges from the top is implied by $\eta \wedge B \wedge 0 \leq E = E_0$; and the weakest precondition for the entire while-statement, which gets written above that while-statement, is $\eta \wedge 0 \leq E$.

Let us illustrate this rule by proving that $\vdash_{\mathsf{tot}} \llparenthesis x \geq 0 \rrparenthesis$ `Fac1` $\llparenthesis y = x! \rrparenthesis$ is valid, where `Fac1` is given in Example 4.2, as follows:

```
y = 1;
z = 0;
while (x != z) {
    z = z + 1;
    y = y * z;
}
```

As already mentioned, $x - z$ is a suitable variant. The invariant $(y = z!)$ of the partial correctness proof is retained. We obtain the following complete proof for total correctness:

```
   $\llparenthesis x \geq 0 \rrparenthesis$
   $\llparenthesis 1 = 0! \wedge 0 \leq x - 0 \rrparenthesis$                                   Implied
y = 1;
   $\llparenthesis y = 0! \wedge 0 \leq x - 0 \rrparenthesis$                                   Assignment
z = 0;
   $\llparenthesis y = z! \wedge 0 \leq x - z \rrparenthesis$                                   Assignment
while (x != z) {
       $\llparenthesis y = z! \wedge x \neq z \wedge 0 \leq x - z = E_0 \rrparenthesis$                 Invariant Hyp. $\wedge$ guard
       $\llparenthesis y \cdot (z+1) = (z+1)! \wedge 0 \leq x - (z+1) < E_0 \rrparenthesis$       Implied
    z = z + 1;
       $\llparenthesis y \cdot z = z! \wedge 0 \leq x - z < E_0 \rrparenthesis$                     Assignment
    y = y * z;
       $\llparenthesis y = z! \wedge 0 \leq x - z < E_0 \rrparenthesis$                         Assignment
}
   $\llparenthesis y = z! \wedge x = z \rrparenthesis$                                          Total-while
   $\llparenthesis y = x! \rrparenthesis$                                                      Implied
```

and so $\vdash_{\mathsf{tot}} \llparenthesis x \geq 0 \rrparenthesis$ `Fac1` $\llparenthesis y = x! \rrparenthesis$ is valid. Two comments are in order:

- Notice that the precondition $x \geq 0$ is crucial in securing the fact that $0 \leq x - z$ holds right before the while-statements gets executed: it implies the precondition $1 = 0! \wedge 0 \leq x - 0$ computed by our proof. In fact, observe that `Fac1` does not terminate if x is negative initially.
- The application of Implied within the body of the while-statement is valid, but it makes vital use of the fact that the boolean guard is true. This is an example of a while-statement whose boolean guard is needed in reasoning about the correctness of *every* iteration of that while-statement.

One may wonder whether there is a program that, given a while-statement and a precondition as input, decides whether that while-statement terminates on all runs whose initial states satisfy that precondition. One can prove that there cannot be such a program. This suggests that the automatic extraction of useful termination expressions E cannot be realized either. Like most other such universal problems discussed in this text, the wish to completely mechanise such decision or extraction procedures cannot be realised. Hence, finding a working variant E is a creative activity which requires skill, intuition and practice.

Let us consider an example program, `Collatz`, that conveys the challenge one may face in finding suitable termination variants E:

```
c = x;
while (c != 1) {
  if (c % 2 == 0) { c = c / 2; }
  else { c = 3*c + 1; }
}
```

This program records the initial value of `x` in `c` and then iterates an if-statement until, and if, the value of `c` equals 1. The if-statement tests whether `c` is even – divisible by 2 – if so, `c` stores its current value divided by 2; if not, `c` stores 'three times its current value plus 1.' The expression `c / 2` denotes integer division, so `11 / 2` renders `5` as does `10 / 2`.

To get a feel for this algorithm, consider an execution trace in which the value of `x` is 5: the value of `c` evolves as `5 16 8 4 2 1`. For another example, if the value of `x` is initially 172, the evolution of `c` is

```
172 86 43 130 65 196 98 49 148 74 37 112 56 28 14 7 22
11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

This execution requires 32 iterations of the while-statement to reach a terminating state in which the value of `c` equals 1. Notice how this trace reaches 5, from where on the continuation is as if 5 were the initial value of `x`.

For the initial value 123456789 of `x` we abstract the evolution of `c` with + (its value increases in the else-branch) and − (its value decreases in the if-branch):

```
+ - - - - - - + - - - + - + - - + - + - + - + - + - + - - + - - -
- + - - - - + - - + - - + - - + - + - - - + - + - - - - - + - - +
- + - - + - - - - + - - - - - - + - - + - + - - + - + - + - - + -
+ - + - + - - + - - - + - + - + - - + - + - - + - + - + - + - + -
+ - - - + - + - + - + - - - - + - - + - - + - - - - + - - - + - +
- + - - - - - + - - - -
```

This requires 177 iterations of the while-statement to reach a terminating state. Although it is re-assuring that some program runs terminate, the irregular pattern of + and − above make it seem very hard, if not impossible, to come up with a variant that proves the termination of `Collatz` on all executions in which the initial value of `x` is positive.

Finally, let's consider a *really big* integer:

```
32498723462509735034567279652376420563047563456356347563\\
96598734085384756074086560785607840745067340563457640875\\
62984573756306537856405634056245634578692825623542135761\\
9519765129854122965424895465956457
```

where \\ denotes concatenation of digits. Although this is a very large number indeed, our program `Collatz` requires only 4940 iterations to terminate. Unfortunately, nobody knows a suitable variant for this program that could prove the validity of $\vdash_{\mathsf{tot}} \llparenthesis\, 0 < x \,\rrparenthesis$ `Collatz` $\llparenthesis\, \top \,\rrparenthesis$. Observe how the use of $\top$ as a postcondition emphasizes that this Hoare triple is merely concerned about program termination as such. Ironically, there is also no known initial value of `x` greater than 0 for which `Collatz` doesn't terminate. In fact, things are even subtler than they may appear: if we replace `3*c + 1` in `Collatz` with a different such linear expression in `c`, the program may not terminate despite meeting the precondition $0 < x$; see exercise 6 on page 303.

4.5 Programming by contract

For a valid sequent $\vdash_{\mathsf{tot}} \llparenthesis\, \phi \,\rrparenthesis\; P \;\llparenthesis\, \psi \,\rrparenthesis$, the triple $\llparenthesis\, \phi \,\rrparenthesis\; P \;\llparenthesis\, \psi \,\rrparenthesis$ may be seen as a *contract* between a supplier and a consumer of a program P. The supplier insists that consumers run P only on initial state satisfies ϕ. In that case, the supplier promises the consumer that the final state of that run satisfies ψ. For a valid $\vdash_{\mathsf{par}} \llparenthesis\, \phi \,\rrparenthesis\; P \;\llparenthesis\, \psi \,\rrparenthesis$, the latter guarantee applies only when a run terminates.

For imperative programming, the validation of Hoare triples can be interpreted as the validation of contracts for method or procedure calls. For example, our program fragment `Fac1` may be the `...` in the method body

```
    int factorial (x: int) { ... return y; }
```

The code for this method can be annotated with its contractual assumptions and guarantees. These annotations can be checked off-line by humans, during compile-time or even at run-time in languages such as Eiffel. A possible format for such contracts for the method `factorial` is given in Figure 4.4.

```
method name:             factorial
input:                   x ofType int
assumes:                 0 <= x
guarantees:              y = x!
output:                  ofType int
modifies only:           y
```

Figure 4.4. A contract for the method `factorial`.

The keyword `assumes` states all preconditions, the keyword `guarantees` lists all postconditions. The keyword `modifies only` specifies which program variables may change their value during an execution of this method.

Let us see why such contracts are useful. Suppose that your boss tells you to write a method that computes $\binom{n}{k}$ – read 'n choose k' – a notion of combinatorics where $1/\binom{49}{6}$ is your change of getting all six lottery numbers right out of 49 numbers total. Your boss also tells you that

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \tag{4.16}$$

holds. The method `factorial` and its contract (Figure 4.4) is at your disposal. Using (4.16) you can quickly compute some values, such as $\binom{5}{2} = 5!/(2! \cdot 3!) = 10$, $\binom{10}{0} = 1$, and $\binom{49}{6} = 13983816$. You then write a method `choose` that makes calls to the method `factorial`, e.g. you may write

```
int choose(n : int, k : int) {
  return factorial(n) / (factorial(k) * factorial (n - k));
}
```

This method body consists of a `return`-statement only which makes three calls to method `factorial` and then computes the result according to (4.16). So far so good. But programming by contract is not just about writing programs, it is also about writing the *contracts* for such programs! The static information about `choose` – e.g. its name – are quickly filled into that contract. But what about the preconditions (`assumes`) and postconditions (`guarantees`)?

At the very least, you must state preconditions that ensure that all method calls within this method's body satisfy *their* preconditions. In this case, we only call `factorial` whose precondition is that its input value be non-negative. Therefore, we require that n, k, and $n - k$ be non-negative. The latter says that n is not smaller than k.

What about the postconditions of `choose`? Since the method body declared no local variables, we use `result` to denote the return value of this method. The postcondition then states that `result` equals $\binom{n}{k}$ – assuming that you boss' equation (4.16) is correct for your preconditions $0 \leq k$, $0 \leq n$, and $k \leq n$. The contract for `choose` is therefore

```
method name:                choose
input:                      n ofType int, k ofType int
assumes:                    0 <= k, 0 <= n, k <= n
guarantees:                 result = ‘n choose k’
output:                     ofType int
modifies only local variables
```

From this we learn that programming by contract uses contracts

1. as assume-guarantee abstract interfaces to methods;
2. to specify their method's header information, output type, when calls to its method are ‘legal,’ what variables that method modifies, and what its output satisfies on all ‘legal’ calls;
3. to enable us to prove the validity of a contract C for method `m` by ensuring that all method calls within `m`'s body meet the preconditions of these methods and using that all such calls then meet their respective postconditions.

Programming by contract therefore gives rise to *program validation by contract.* One proves the ‘Hoare triple’ $\llparenthesis \texttt{assume} \rrparenthesis\ \texttt{method}\ \llparenthesis \texttt{guarantee} \rrparenthesis$ very much in the style developed in this chapter, except that for all method invocations within that body we can assume that *their* Hoare triples are correct.

Example 4.21 We have already used program validation by contract in our verification of the program that computes the minimal sum for all sections of an array in Figure 4.3 on page 291. Let us focus on the proof fragment

```
$\llparenthesis \texttt{Inv1}(\min(s, \min(t + a[k], a[k])), k + 1) \wedge \texttt{Inv2}(\min(t + a[k], a[k]), k + 1) \rrparenthesis$
                                                          Implied (Lemma 4.20)
    t = min(t + a[k], a[k]);
      $\llparenthesis \texttt{Inv1}(\min(s, t), k + 1) \wedge \texttt{Inv2}(t, k + 1) \rrparenthesis$          Assignment
    s = min(s,t);
      $\llparenthesis \texttt{Inv1}(s, k + 1) \wedge \texttt{Inv2}(t, k + 1) \rrparenthesis$                  Assignment
```

Its last line serves as the postcondition which gets pushed through the assignment `s = min(s,t)`. But `min(s,t)` is a method call whose guarantees are specified as ‘`result` equals $\min(s, t)$,’ where $\min(s, t)$ is a mathematical notation for the smaller of the numbers s and t. Thus, the rule Assignment does not substitute the syntax of the method invocation `min(s,t)` for all occurrences of s in $\texttt{Inv1}(s, k + 1) \wedge \texttt{Inv2}(t, k + 1)$, but changes all such s to the guarantee $\min(s, t)$ of the method call `min(s,t)` – program validation by contract in action! A similar comment applies for the assignment `t = min(t + a[k], a[k])`.

Program validation by contract has to be used wisely to avoid circular reasoning. If each method is a node in a graph, let's draw an edge from method `n` to method `m` iff within the body of `n` there is a call to method `m`. For program validation by contract to be sound, we require that there be no cycles in this method-dependency graph.

4.6 Exercises

Exercises 4.1

* 1. If you already have written computer programs yourself, assemble for each programming language you used a list of features of its software development environment (compiler, editor, linker, run-time environment etc) that may improve the likelihood that your programs work correctly. Try to rate the effectiveness of each such feature.

2. Repeat the previous exercise by listing and rating features that may decrease the likelihood of procuding correct and reliable programs.

——

Exercises 4.2

* 1. In what circumstances would `if` (B) $\{C_1\}$ `else` $\{C_2\}$ fail to terminate?

* 2. A familiar command missing from our language is the for-statement. It may be used to sum the elements in an array, for example, by programming as follows:

```
s = 0;
for (i = 0; i <= max; i = i+1) {
   s = s + a[i];
}
```

After performing the initial assignment `s = 0`, this executes `i = 0` first, then executes the body `s = s + a[i]` and the incrementation `i = i + 1` continually until `i <= max` becomes false. Explain how `for` $(C_1; B; C_2)$ $\{C_3\}$ can be defined as a derived program in our core language.

3. Suppose that you need a language construct `repeat` $\{C\}$ `until` (B) which repeats C until B becomes true, i.e.

   i. executes C in the current state of the store;

   ii. evaluates B in the resulting state of the store;

   iii. if B is false, the program resumes with (i); otherwise, the program `repeat` $\{C\}$ `until` (B) terminates.

   This construct sometimes allows more elegant code than a corresponding while-statement.

(a) Define `repeat` C `until` B as a derived expression using our core language.

(b) Can one define every `repeat` expression in our core language extended with for-statements? (You might need the empty command `skip` which does nothing.)

Exercises 4.3

1. For any store l as in Example 4.4 (page 264), determine which of the relations below hold; justify your answers:
   * (a) $l \vDash (x + y < z) \to \neg(x * y = z)$
   (b) $l \vDash \forall u\,(u < y) \lor (u * z < y * z)$
   * (c) $l \vDash x + y - z < x * y * z$.
* 2. For any ϕ, ψ and P explain why $\vDash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds whenever the relation $\vDash_{\mathsf{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds.
3. Let the relation $P \vdash l \leadsto l'$ hold iff P's execution in store l terminates, resulting in store l'. Use this formal judgment $P \vdash l \leadsto l'$ along with the relation $l \vDash \phi$ to define $\vDash_{\mathsf{par}}$ and $\vDash_{\mathsf{tot}}$ symbolically.
4. Another reason for proving partial correctness in isolation is that some program fragments have the form `while (true) {C}`. Give useful examples of such program fragments in application programming.
* 5. Use the proof rule for assignment and logical implication as appropriate to show the validity of
   (a) $\vdash_{\mathsf{par}} (\!|x > 0|\!)$ `y = x + 1` $(\!|y > 1|\!)$
   (b) $\vdash_{\mathsf{par}} (\!|\top|\!)$ `y = x; y = x + x + y` $(\!|y = 3 \cdot x|\!)$
   (c) $\vdash_{\mathsf{par}} (\!|x > 1|\!)$ `a = 1; y = x; y = y - a` $(\!|y > 0 \land x > y|\!)$.
* 6. Write down a program P such that
   (a) $(\!|\top|\!)\ P\ (\!|y = x + 2|\!)$
   (b) $(\!|\top|\!)\ P\ (\!|z > x + y + 4|\!)$
   holds under partial correctness; then prove that this is so.
7. For all instances of Implied in the proof on page 274, specify their corresponding $\vdash_{\mathrm{AR}}$ sequents.
8. There is a safe way of relaxing the format of the proof rule for assignment: as long as no variable occurring in E gets updated in between the assertion $\psi[E/x]$ and the assignment `x = E` we may conclude ψ right after this assignment. Explain why such a proof rule is sound.
9. (a) Show, by means of an example, that the 'reversed' version of the rule Implied

$$\frac{\vdash_{\mathrm{AR}} \phi \to \phi' \qquad (\!|\phi|\!)\ C\ (\!|\psi|\!) \qquad \vdash_{\mathrm{AR}} \psi' \to \psi}{(\!|\phi'|\!)\ C\ (\!|\psi'|\!)}\ \text{Implied\_Reversed}$$

   is unsound for partial correctness.
   (b) Explain why the modified rule If-Statement in (4.7) is sound with respect to the partial and total satisfaction relation.

* (c) Show that any instance of the modified rule If-Statement in a proof can be replaced by an instance of the original If-statement and instances of the rule Implied. Is the converse true as well?

* 10. Prove the validity of the sequent $\vdash_{\mathsf{par}} \llparenthesis \top \rrparenthesis\ P\ \llparenthesis z = \min(x, y) \rrparenthesis$, where $\min(x, y)$ is the smallest number of x and y – e.g. $\min(7, 3) = 3$ – and the code of P is given by

```
if (x > y) {
   z = y;
} else {
   z = x;
}
```

11. For each of the specifications below, write code for P and prove the partial correctness of the specified input/output behaviour:

* (a) $\llparenthesis \top \rrparenthesis\ P\ \llparenthesis z = \max(w, x, y) \rrparenthesis$, where $\max(w, x, y)$ denotes the largest of w, x and y.

* (b) $\llparenthesis \top \rrparenthesis\ P\ \llparenthesis ((x = 5) \to (y = 3)) \land ((x = 3) \to (y = -1)) \rrparenthesis$.

12. Prove the validity of the sequent $\vdash_{\mathsf{par}} \llparenthesis \top \rrparenthesis$ `Succ` $\llparenthesis y = x + 1 \rrparenthesis$ without using the modified proof rule for if-statements.

* 13. Show that $\vdash_{\mathsf{par}} \llparenthesis x \geq 0 \rrparenthesis$ `Copy1` $\llparenthesis x = y \rrparenthesis$ is valid, where `Copy1` denotes the code

```
a = x;
y = 0;
while (a != 0) {
   y = y + 1;
   a = a - 1;
}
```

* 14. Show that $\vdash_{\mathsf{par}} \llparenthesis y \geq 0 \rrparenthesis$ `Multi1` $\llparenthesis z = x \cdot y \rrparenthesis$ is valid, where `Multi1` is:

```
a = 0;
z = 0;
while (a != y) {
  z = z + x;
  a = a + 1;
}
```

15. Show that $\vdash_{\mathsf{par}} \llparenthesis y = y_0 \land y \geq 0 \rrparenthesis$ `Multi2` $\llparenthesis z = x \cdot y_0 \rrparenthesis$ is valid, where `Multi2` is:

```
z = 0;
while (y != 0) {
  z = z + x;
  y = y - 1;
}
```

16. Show that $\vdash_{\mathsf{par}} \llparenthesis x \geq 0 \rrparenthesis$ `Copy2` $\llparenthesis x = y \rrparenthesis$ is valid, where `Copy2` is:

```
y = 0;
while (y != x) {
  y = y + 1;
}
```

17. The program Div is supposed to compute the dividend of integers x by y; this is defined to be the unique integer d such that there exists some integer r – the remainder – with $r < y$ and $x = d \cdot y + r$. For example, if $x = 15$ and $y = 6$, then $d = 2$ because $15 = 2 \cdot 6 + 3$, where $r = 3 < 6$. Let Div be given by:

```
r = x;
d = 0;
while (r >= y) {
  r = r - y;
  d = d + 1;
}
```

Show that $\vdash_{\mathsf{par}} (\!| \neg(y = 0) |\!)$ Div $(\!| (x = d \cdot y + r) \wedge (r < y) |\!)$ is valid.

* 18. Show that $\vdash_{\mathsf{par}} (\!| x \geq 0 |\!)$ Downfac $(\!| y = x! |\!)$ is valid[7], where Downfac is:

```
a = x;
y = 1;
while (a > 0) {
  y = y * a;
  a = a - 1;
}
```

19. Why can, or can't, you prove the validity of $\vdash_{\mathsf{par}} (\!| \top |\!)$ Copy1 $(\!| x = y |\!)$?
20. Let all while-statements while (B) {C} in P be annotated with invariant candidates η at the and of their bodies, and $\eta \wedge B$ at the beginning of their body.
    (a) Explain how a proof of $\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ can be automatically reduced to showing the validity of some $\vdash_{\mathrm{AR}} \psi_1 \wedge \cdots \wedge \psi_n$.
    (b) Identify such a sequent $\vdash_{\mathrm{AR}} \psi_1 \wedge \cdots \wedge \psi_n$ for the proof in Example 4.17 on page 287.
21. Given $n = 5$ test the correctness of Min_Sum on the arrays below:
    * (a) $[-3, 1, -2, 1, -8]$
    (b) $[1, 45, -1, 23, -1]$
    * (c) $[-1, -2, -3, -4, 1097]$.
22. If we swap the first and second assignment in the while-statement of Min_Sum, so that it first assigns to s and then to t, is the program still correct? Justify your answer.

* 23. Prove the partial correctness of **S2** for Min_Sum.

24. The program Min_Sum does not reveal where a minimal-sum section may be found in an input array. Adapt Min_Sum to achieve that. Can you do this with a single pass through the array?
25. Consider the proof rule

$$\frac{(\!| \phi |\!)\ C\ (\!| \psi_1 |\!) \qquad (\!| \phi |\!)\ C\ (\!| \psi_2 |\!)}{(\!| \phi |\!)\ C\ (\!| \psi_1 \wedge \psi_2 |\!)}\ \mathsf{Conj}$$

[7] You may have to strengthen your invariant.

for Hoare triples.

(a) Show that this proof rule is sound for $\vDash_{\mathsf{par}}$.

(b) Derive this proof rule from the ones on page 270.

(c) Explain how this rule, or its derived version, is used to establish the overall correctness of `Min_Sum`.

26. The maximal-sum problem is to compute the maximal sum of all sections on an array.

(a) Adapt the program from page 289 so that it computes the maximal sum of these sections.

(b) Prove the partial correctess of your modified program.

(c) Which aspects of the correctness proof given in Figure 4.3 (page 291) can be 're-used?'

Exercises 4.4

1. Prove the validity of the following total-correctness sequents:

* (a) $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Copy1` $(\!| x = y |\!)$

* (b) $\vdash_{\mathsf{tot}} (\!| y \geq 0 |\!)$ `Multi1` $(\!| z = x \cdot y |\!)$

(c) $\vdash_{\mathsf{tot}} (\!| (y = y_0) \wedge (y \geq 0) |\!)$ `Multi2` $(\!| z = x \cdot y_0 |\!)$

* (d) $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Downfac` $(\!| y = x! |\!)$

* (e) $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Copy2` $(\!| x = y |\!)$, does your invariant have an active part in securing correctness?

(f) $\vdash_{\mathsf{tot}} (\!| \neg(y = 0) |\!)$ `Div` $(\!| (x = d \cdot y + r) \wedge (r < y) |\!)$.

2. Prove total correctness of **S1** and **S2** for `Min_Sum`.
3. Prove that $\vdash_{\mathsf{par}}$ is sound for $\vDash_{\mathsf{par}}$. Just like in Section 1.4.3, it suffices to assume that the premises of proof rules are instances of $\vDash_{\mathsf{par}}$. Then, you need to prove that their respective conclusion must be an instance of $\vDash_{\mathsf{par}}$ as well.
4. Prove that $\vdash_{\mathsf{tot}}$ is sound for $\vDash_{\mathsf{tot}}$.
5. Implement program `Collatz` in a programming language of your choice such that the value of `x` is the program's input and the final value of `c` its output. Test your program on a range of inputs. Which is the biggest integer for which your program terminates without raising an exception or dumping the core?
6. A function over integers $f \colon \mathbb{I} \to \mathbb{I}$ is affine iff there are integers a and b such that $f(x) = a \cdot x + b$ for all $x \in \mathbb{I}$. The else-branch of the program `Collatz` assigns to `c` the value $f(c)$, where f is an affine function with $a = 3$ and $b = 1$.

(a) Write an parameterized implementation of `Collatz` in which you can initially specify the values of a and b either statically or through keyboard input such that the else-branch assigns to `c` the value of $f(c)$.

(b) Determine for which pairs $(a, b) \in \mathbb{I} \times \mathbb{I}$ the set $\mathrm{Pos} \stackrel{\mathrm{def}}{=} \{x \in \mathbb{I} \mid 0 < x\}$ is invariant under the affine function $f(x) = a \cdot x + b$: for all $x \in \mathrm{Pos}$, $f(x) \in \mathrm{Pos}$.

* (c) Find an affine function that leaves Pos invariant, but not the set $\mathrm{Odd} \stackrel{\mathrm{def}}{=} \{x \in \mathbb{I} \mid \exists y \in \mathbb{I} \colon x = 2 \cdot y + 1\}$, such that there is an input drawn from Pos whose

execution with the modified `Collatz` program eventually enters a cycle, and therefore does not terminate.

——

Exercises 4.5

1. Consider methods of the form `boolean certify_V(c : Certificate)` which return `true` iff the certificate `c` is judged valid by the verifier `V`, a class in which method `certify_V` resides.

   * (a) Discuss how programming by contract can be used to delegate the judgment of a certificate to another verifier.

   * (b) What potential problems do you see in this context if the resulting method-dependency graph is circular?

* 2. Consider the method

```
boolean withdraw(amount: int) {
  if (amount < 0 && isGood(amount))
     { balance = balance - amount;
        return true;
     } else { return false; }
}
```

named `withdraw` which attempts to withdraw `amount` from an integer field `balance` of the class within which method `withdraw` lives. This method makes use of another method `isGood` which returns `true` iff the value of `balance` is greater or equal to the value of `amount`.

(a) Write a contract for method `isGood`.

(b) Use that contract to show the validity of the contract for `withdraw`:

```
method name:                  withdraw
input:                        amount of Type int
assumes:                      0 <= balance
guarantees:                   0 <= balance
output:                       of Type boolean
modifies only:                balance
```

Notice that the precondition and postcondition of this contract are the same and refer to a field of the method's object. Upon validation, this contract establishes that all calls to `withdraw` leave (the 'object invariant') `0 <= balance` invariant.

——

4.7 Bibliographic notes

An early exposition of the program logics for partial and total correctness of programs written in an imperative while-language can be found in [Hoa69]. The text [Dij76] contains a formal treatment of weakest preconditions.

Backhouse's book [Bac86] describes program logic and weakest preconditions and also contains numerous examples and exercises. Other books giving more complete expositions of program verification than we can in this chapter are [AO91, Fra92]; they also extend the basic core language to include features such as procedures and parallelism. The issue of writing to arrays and the problem of array cell aliasing are described in [Fra92]. The original article describing the minimal-sum section problem is [Gri82]. A gentle introduction to the mathematical foundations of functional programming is [Tur91]. Some web sites deal with software liability and possible standards for intellectual property rights applied to computer programs[8] [9]. Text books on systematic programming language design by uniform extensions of the core language we presented at the beginning of this chapter are [Ten91, Sch94]. A text on functional programming on the freely available language Standard ML of New Jersey is [Pau91].

[8] `www.opensource.org`

[9] `www.sims.berkeley.edu/~pam/papers.html`

5
Modal logics and agents

5.1 Modes of truth

In propositional or predicate logic, formulas are either true, or false, in any model. Propositional logic and predicate logic do not allow for any further possibilities. From many points of view, however, this is inadequate. In natural language, for example, we often distinguish between various 'modes' of truth, such as *necessarily true*, *known to be true*, *believed to be true* and *true in the future*. For example, we would say that, although the sentence

George W. Bush is president of the United States of America.

is currently true, it will not be true at some point in the future. Equally, the sentence

There are nine planets in the solar system.

while true, and maybe true for ever in the future, is not necessarily true, in the sense that it could have been a different number. However, the sentence

The cube root of 27 is 3.

as well as being true is also necessarily true and true in the future. It does not enjoy all modes of truth, however. It may not be known to be true by some people (children, for example); it may not be believed by others (if they are mistaken).

In computer science, it is often useful to reason about modes of truth. In Chapter 3, we studied the logic CTL in which we could distinguish not only between truth at different points in the future, but also between different futures. Temporal logic is thus a special case of modal logic. The modalities of CTL allow us to express a host of computational behaviour of systems. Modalities are also extremely useful in modelling other domains of computer science. In artificial intelligence, for example, scenarios with several interacting agents are developed. Each agent may have different knowledge about the environment and also about the knowledge of other agents. In this chapter, we will look in depth at modal logics applied to reasoning about knowledge.

Modal logic adds unary connectives to express one, or more, of these different modes of truth. The simplest modal logics just deal with one concept – such as knowledge, necessity, or time. More sophisticated modal logics have connectives for expressing several modes of truth in the same logic; we will see some of these towards the end of this chapter.

We take a *logic engineering* approach in this chapter, in which we address the following question: given a particular mode of truth, how may we develop a logic capable of expressing and formalising that concept? To answer this question, we need to decide what properties the logic should have and what examples of reasoning it should be able to express. Our main case study will be the logic of *knowledge in a multi-agent system*. But first, we look at the syntax and semantics of basic modal logic.

5.2 Basic modal logic

5.2.1 Syntax

The language of basic modal logic is that of propositional logic with two extra connectives, $\Box$ and $\Diamond$. Like negation ($\neg$), they are *unary* connectives as they apply themselves to a single formula only. As done in Chapters 1 and 3, we write $p, q, r, p_3 \ldots$ to denote atomic formulas.

Definition 5.1 The formulas of basic modal logic ϕ are defined by the following Backus Naur form (BNF):

$$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid (\phi \leftrightarrow \phi) \mid (\Box\phi) \mid (\Diamond\phi) \tag{5.1}$$

where p is any atomic formula.

Example formulas of basic modal logic are $(p \wedge \Diamond(p \to \Box\neg r))$ and $\Box((\Diamond q \wedge \neg r) \to \Box p)$, having the parse trees shown in Figure 5.1. The following strings are *not* formulas, because they cannot be constructed using the grammar in (5.1): $(p\Box \to q)$ and $(p \to \Diamond(q \Diamond r))$.

Convention 5.2 As done in Chapter 1, we assume that the unary connectives ($\neg$, $\Box$ and $\Diamond$) bind most closely, followed by $\wedge$ and $\vee$ and then followed by $\to$ and $\leftrightarrow$.

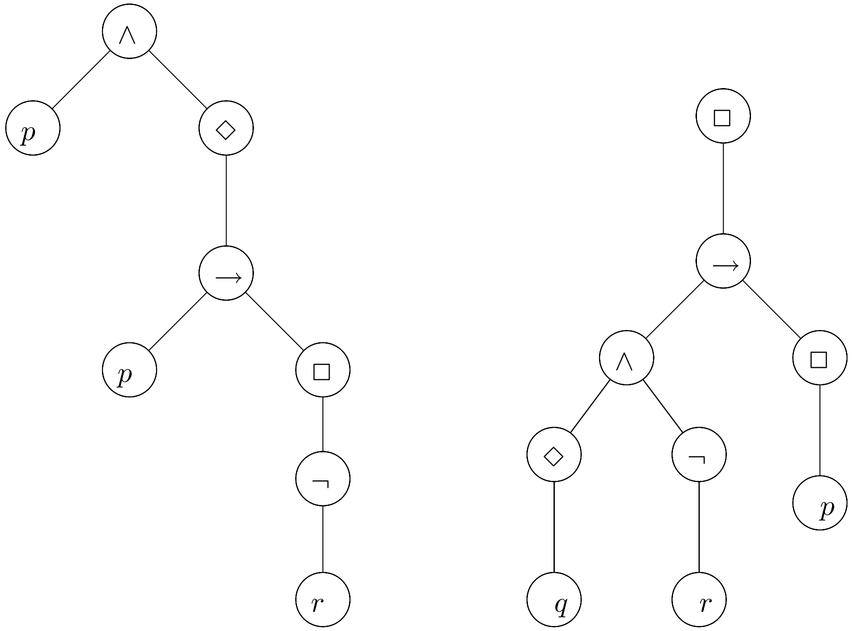


Figure 5.1. Parse trees for $(p \wedge \Diamond(p \rightarrow \Box\neg r))$ and $\Box((\Diamond q \wedge \neg r) \rightarrow \Box p)$.

This convention allows us to remove many sets of brackets, retaining them only to avoid ambiguity, or to override these binding priorities. For example, $\Box((\Diamond q \wedge \neg r) \rightarrow \Box p)$ can be written $\Box(\Diamond q \wedge \neg r \rightarrow \Box p)$. We cannot omit the remaining brackets, however, for $\Box\Diamond q \wedge \neg r \rightarrow \Box p$ has quite a different parse tree (see Figure 5.2) from the one in Figure 5.1.

In basic modal logic, $\Box$ and $\Diamond$ are read 'box' and 'diamond,' but, when we apply modal logics to express various modes of truth, we may read them appropriately. For example, in the logic that studies necessity and possibility, $\Box$ is read 'necessarily' and $\Diamond$ 'possibly;' in the logic of agent Q's knowledge, $\Box$ is read 'agent Q knows' and $\Diamond$ is read 'it is consistent with agent Q's knowledge that,' or more colloquially, 'for all Q knows.' We will see why these readings are appropriate later in the chapter.

5.2.2 Semantics

For a formula of propositional logic, a model is simply an assignment of truth values to each of the atomic formulas present in that formula – we called such models valuation in Chapter 1. However, this notion of model is inadequate for modal logic, since we want to distinguish between different modes, or degrees, of truth.

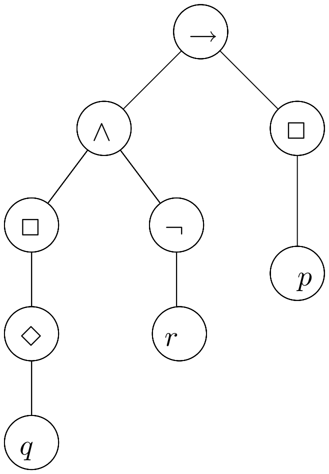


Figure 5.2. The parse tree for $\Box\Diamond q \land \neg r \to \Box p$.

Definition 5.3 A model $\mathcal{M}$ of basic modal logic is specified by three things:

1. A set W, whose elements are called worlds;
2. A relation R on W ($R \subseteq W \times W$), called the accessibility relation;
3. A function $L : W \to \mathcal{P}(\texttt{Atoms})$, called the labelling function.

We write $R(x, y)$ to denote that (x, y) is in R.

These models are often called *Kripke models*, in honour of S. Kripke who invented them and worked extensively in modal logic in the 1950s and 1960s. Intuitively, $w \in W$ stands for a possible world and $R(w, w')$ means that w' is a world *accessible from* world w. The actual nature of that relationship depends on what we intend to model. Although the definition of models looks quite complicated, we can use an easy graphical notation to depict finite models. We illustrate the graphical notation by an example. Suppose W equals $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ and the relation R is given as follows:

- $R(x_1, x_2)$, $R(x_1, x_3)$, $R(x_2, x_2)$, $R(x_2, x_3)$, $R(x_3, x_2)$, $R(x_4, x_5)$, $R(x_5, x_4)$, $R(x_5, x_6)$; and no other pairs are related by R.

Suppose further that the labelling function behaves as follows:

x	x_1	x_2	x_3	x_4	x_5	x_6
$L(x)$	$\{q\}$	$\{p, q\}$	$\{p\}$	$\{q\}$	$\emptyset$	$\{p\}$

Then, the Kripke model is illustrated in Figure 5.3. The set W is drawn as a set of circles, with arrows between them showing the relation R. Within each circle is the value of the labelling function in that world. If you have read Chapter 3, then you might have noticed that Kripke structures are also the models for CTL, where W is S, the set of states; R is $\longrightarrow$, the relation of state transitions; and L is the labelling function.

Definition 5.4 Let $\mathcal{M} = (W, R, L)$ be a model of basic modal logic. Suppose $x \in W$ and ϕ is a formula of (5.1). We will define when formula ϕ is true in the world x. This is done via a satisfaction relation $x \Vdash \phi$ by structural induction on ϕ:

$$\begin{array}{rl}
x \Vdash \top & \\
x \nVdash \bot & \\
x \Vdash p & \text{iff } p \in L(x) \\
x \Vdash \neg\phi & \text{iff } x \nVdash \phi \\
x \Vdash \phi \wedge \psi & \text{iff } x \Vdash \phi \text{ and } x \Vdash \psi \\
x \Vdash \phi \vee \psi & \text{iff } x \Vdash \phi\text{ , or } x \Vdash \psi \\
x \Vdash \phi \rightarrow \psi & \text{iff } x \Vdash \psi\text{ , whenever we have } x \Vdash \phi \\
x \Vdash \phi \leftrightarrow \psi & \text{iff } (x \Vdash \phi \text{ iff } x \Vdash \psi) \\
x \Vdash \Box\psi & \text{iff, for each } y \in W \text{ with } R(x, y)\text{, we have } y \Vdash \psi \\
x \Vdash \Diamond\psi & \text{iff there is a } y \in W \text{ such that } R(x, y) \text{ and } y \Vdash \psi.
\end{array}$$

When $x \Vdash \phi$ holds, we say 'x satisfies ϕ,' or 'ϕ is true in world x.' We write $\mathcal{M}, x \Vdash \phi$ if we want to stress that $x \Vdash \phi$ holds in the model $\mathcal{M}$.

The first two clauses just express the fact that $\top$ is always true, while $\bot$ is always false. Next, we see that $L(x)$ is the set of all the atomic formulas that are true at x. The clauses for the boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$) should also be straightforward: they mean that we apply the usual truth-table semantics of these connectives in the current world x. The interesting cases are those for $\Box$ and $\Diamond$. For $\Box\phi$ to be true at x, we require that ϕ be true in all the worlds accessible by R from x. For $\Diamond\phi$, it is required that there is at least one accessible world in which ϕ is true. Thus, $\Box$ and $\Diamond$ are a bit like the quantifiers $\forall$ and $\exists$ of predicate logic, except that they do not take variables as arguments. This fact makes them conceptually much simpler than quantifiers. The modal operators $\Box$ and $\Diamond$ are also rather like AX and EX in CTL – see Section 3.4.1. Note that the meaning of $\phi_1 \leftrightarrow \phi_2$ coincides with that of $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$; we call it 'if and only if.'

Definition 5.5 A model $\mathcal{M} = (W, R, L)$ of basic modal logic is said to satisfy a formula if every state in the model satisfies it. Thus, we write $\mathcal{M} \vDash \phi$ iff, for each $x \in W$, $x \Vdash \phi$.

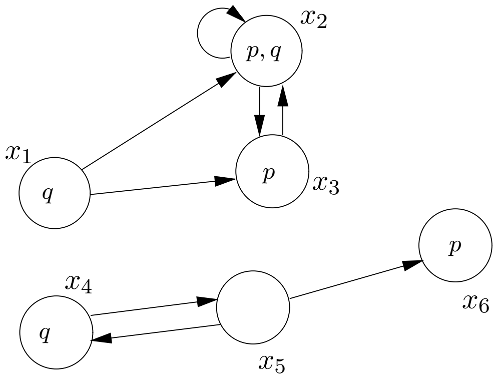


Figure 5.3. A Kripke model.

Examples 5.6 Consider the Kripke model of Figure 5.3. We have:

- $x_1 \Vdash q$, since $q \in L(x_1)$.
- $x_1 \Vdash \Diamond q$, for there is a world accessible from x_1 (namely, x_2) which satisfies q. In mathematical notation: $R(x_1, x_2)$ and $x_2 \Vdash q$.
- $x_1 \nVdash \Box q$, however. This is because $x_1 \Vdash \Box q$ says that all worlds accessible from x_1 (i.e. x_2 and x_3) satisfy q; but x_3 does not.
- $x_5 \nVdash \Box p$ and $x_5 \nVdash \Box q$. Moreover, $x_5 \nVdash \Box p \vee \Box q$. However, $x_5 \Vdash \Box(p \vee q)$.
  To see these facts, note that the worlds accessible from x_5 are x_4 and x_6. Since $x_4 \nVdash p$, we have $x_5 \nVdash \Box p$; and since $x_6 \nVdash q$, we have $x_5 \nVdash \Box q$. Therefore, we get that $x_5 \nVdash \Box p \vee \Box q$. However, $x_5 \Vdash \Box(p \vee q)$ holds because, in each of x_4 and x_6, we find p or q.
- The worlds which satisfy $\Box p \to p$ are x_2, x_3, x_4, x_5 and x_6; for x_2, x_3 and x_6 this is so since they already satisfy p; for x_4 this is true since it does not satisfy $\Box p$ – we have $R(x_4, x_5)$ and x_5 does not satisfy p; a similar reason applies to x_5. As for x_1, it cannot satisfy $\Box p \to p$ since it satisfies $\Box p$ but not p itself.

Worlds like x_6 that have no world accessible to them deserve special attention in modal logic. Observe that $x_6 \nVdash \Diamond \phi$, no matter what ϕ is, because $\Diamond \phi$ says 'there is an accessible world which satisfies ϕ.' In particular, 'there is an accessible world,' which in the case of x_6 there is not. Even when ϕ is $\top$, we have $x_6 \nVdash \Diamond \top$. So, although $\top$ is satisfied in every world, $\Diamond \top$ is not necessarily. In fact, $x \Vdash \Diamond \top$ holds iff x has at least one accessible world.

A dual situation exists for the satisfaction of $\Box \phi$ in worlds with no accessible world. No matter what ϕ is, we find that $x_6 \Vdash \Box \phi$ holds. That is because $x_6 \Vdash \Box \phi$ says that ϕ is true in all worlds accessible from x_6. There are no such worlds, so ϕ is vacuously true in all of them: there is simply nothing to check. This reading of 'for all accessible worlds' may seem surprising, but it secures the de Morgan rules for the box and diamond modalities shown

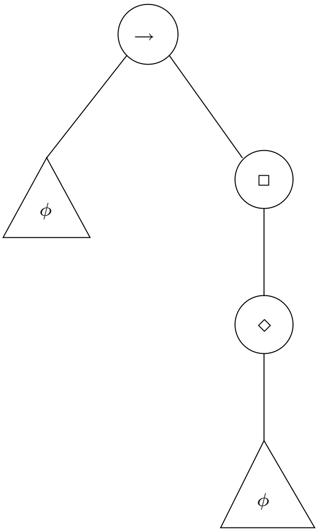


Figure 5.4. The parse tree of the formula scheme $\phi \to \Box\Diamond\phi$.

below. Even $\Box\bot$ is true in x_6. If you wanted to convince someone that $\Box\bot$ was not true in x_6, you'd have to show that there is a world accessible from x_6 in which $\bot$ is not true; but you can't do this, for there are no worlds accessible from x_6. So again, although $\bot$ is false in every world, $\Box\bot$ might not be false. In fact, $x \Vdash \Box\bot$ holds iff x has no accessible worlds.

Formulas and formula schemes The grammar in (5.1) specifies exactly the formulas of basic modal logic, given a set of atomic formulas. For example, $p \to \Box\Diamond p$ is such a formula. It is sometimes useful to talk about a whole family of formulas which have the same 'shape;' these are called *formula schemes*. For example, $\phi \to \Box\Diamond\phi$ is a formula scheme. Any formula which has the shape of a certain formula scheme is called an *instance* of the scheme. For example,

- $p \to \Box\Diamond p$
- $q \to \Box\Diamond q$
- $(p \wedge \Diamond q) \to \Box\Diamond(p \wedge \Diamond q)$

are all instances of the scheme $\phi \to \Box\Diamond\phi$. An example of a formula scheme of propositional logic is $\phi \wedge \psi \to \psi$. We may think of a formula scheme as an under-specified parse tree, where certain portions of the tree still need to be supplied – e.g. the tree of $\phi \to \Box\Diamond\phi$ is found in Figure 5.4.

Semantically, a scheme can be thought of as the conjunction of all its instances – since there are generally infinitely many such instances, this cannot be carried out syntactically! We say that a world/model satisfies a scheme if it satisfies all its instances. Note that an instance being satisfied in a Kripke model does not imply that the whole scheme is satisfied. For example, we may have a Kripke model in which all worlds satisfy $\neg p \vee q$, but at least one world does not satisfy $\neg q \vee p$; the scheme $\neg\phi \vee \psi$ is not satisfied.

Equivalences between modal formulas

Definition 5.7 1. We say that a set of formulas Γ of basic modal logic semantically entails a formula ψ of basic modal logic if, in any world x of any model $\mathcal{M} = (W, R, L)$, we have $x \Vdash \psi$ whenever $x \Vdash \phi$ for all $\phi \in \Gamma$. In that case, we say that $\Gamma \vDash \psi$ holds.

2. We say that ϕ and ψ are semantically equivalent if $\phi \vDash \psi$ and $\psi \vDash \phi$ hold. We denote this by $\phi \equiv \psi$.

Note that $\phi \equiv \psi$ holds iff any world in any model which satisfies one of them also satisfies the other. The definition of semantic equivalence is based on semantic entailment in the same way as the corresponding one for formulas of propositional logic. However, the underlying notion of semantic entailment for modal logic is quite different, as we will see shortly.

Any equivalence in propositional logic is also an equivalence in modal logic. Indeed, if we take any equivalence in propositional logic and substitute the atoms uniformly for any modal logic formula, the result is also an equivalence in modal logic. For example, take the equivalent formulas $p \to \neg q$ and $\neg(p \wedge q)$ and now perform the substitution

$$\begin{array}{rcl} p & \mapsto & \Box p \wedge (q \to p) \\ q & \mapsto & r \to \Diamond(q \vee p). \end{array}$$

The result of this substitution is the pair of formulas

$$\begin{array}{c} \Box p \wedge (q \to p) \to \neg(r \to \Diamond(q \vee p)) \\ \neg((\Box p \wedge (q \to p)) \wedge (r \to \Diamond(q \vee p))) \end{array} \tag{5.2}$$

which are equivalent as formulas of basic modal logic.

We have already noticed that $\Box$ is a universal quantifier on accessible worlds and $\Diamond$ is the corresponding existential quantifier. In view of these facts, it is not surprising to find that de Morgan rules apply for $\Box$ and $\Diamond$:

$$\neg\Box\phi \equiv \Diamond\neg\phi \text{ and } \neg\Diamond\phi \equiv \Box\neg\phi.$$

Moreover, $\Box$ distributes over $\wedge$ and $\Diamond$ distributes over $\vee$:

$$\Box(\phi \wedge \psi) \equiv \Box\phi \wedge \Box\psi \text{ and } \Diamond(\phi \vee \psi) \equiv \Diamond\phi \vee \Diamond\psi.$$

These equivalences correspond closely to the quantifier equivalences discussed in Section 2.3.2. It is also not surprising to find that $\Box$ does *not* distribute over $\vee$ and $\Diamond$ does *not* distribute over $\wedge$, i.e. we do not have equivalences between $\Box(\phi \vee \psi)$ and $\Box\phi \vee \Box\psi$, or between $\Diamond(\phi \wedge \psi)$ and $\Diamond\phi \wedge \Diamond\psi$. For example, in the fourth item of Example 5.6 we had $x_5 \Vdash \Box(p \vee q)$ and $x_5 \nVdash \Box p \vee \Box q$.

Note that $\Box\top$ is equivalent to $\top$, but *not* to $\Diamond\top$, as we saw earlier. Similarly, $\Diamond\bot \equiv \bot$ but they are not equivalent to $\Box\bot$.

Another equivalence is $\Diamond\top \equiv \Box p \to \Diamond p$. For suppose $x \Vdash \Diamond\top$ – i.e. x has an accessible world, say y – and suppose $x \Vdash \Box p$; then $y \Vdash p$, so $x \Vdash \Diamond p$. Conversely, suppose $x \Vdash \Box p \to \Diamond p$; we must show it satisfies $\Diamond\top$. Let us distinguish between the cases $x \Vdash \Box p$ and $x \nVdash \Box p$; in the former, we get $x \Vdash \Diamond p$ from $x \Vdash \Box p \to \Diamond p$ and so x must have an accessible world; and in the latter, x must again have an accessible world in order to avoid satisfying $\Box p$. Either way, x has an accessible world, i.e. satisfies $\Diamond\top$. Naturally, this argument works for any formula ϕ, not just an atom p.

Valid formulas

Definition 5.8 A formula ϕ of basic modal logic is said to be valid if it is true in every world of every model, i.e. iff $\vDash \phi$ holds.

Any propositional tautology is a valid formula and so is any substitution instance of it. A substitution instance of a formula is the result of uniformly substituting the atoms of the formula by other formulas as done in (5.2). For example, since $p \vee \neg p$ is a tautology, performing the substitution $p \mapsto \Box p \wedge (q \to p)$ gives us a valid formula $(\Box p \wedge (q \to p)) \vee \neg(\Box p \wedge (q \to p))$.

As we may expect from equivalences above, these formulas are valid:

$$\begin{aligned} &\neg\Box\phi \leftrightarrow \Diamond\neg\phi \\ &\Box(\phi \wedge \psi) \leftrightarrow \Box\phi \wedge \Box\psi \\ &\Diamond(\phi \vee \psi) \leftrightarrow \Diamond\phi \vee \Diamond\psi. \end{aligned} \tag{5.3}$$

To prove that the first of these is valid, we reason as follows. Suppose x is a world in a model $\mathcal{M} = (W, R, L)$. We want to show $x \Vdash \neg\Box\phi \leftrightarrow \Diamond\neg\phi$, i.e. that $x \Vdash \neg\Box\phi$ iff $x \Vdash \Diamond\neg\phi$. Well, using Definition 5.4,

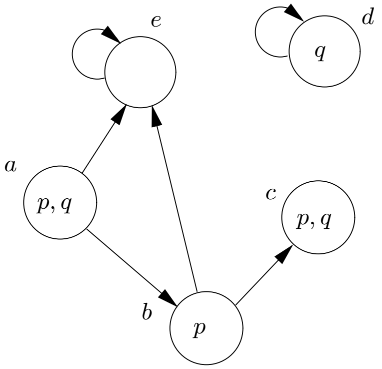


Figure 5.5. Another Kripke model.

$x \Vdash \neg\Box\phi$
    iff it isn't the case that $x \Vdash \Box\phi$
    iff it isn't the case that, for all y such that $R(x, y)$, $y \Vdash \phi$
    iff there is some y such that $R(x, y)$ and not $y \Vdash \phi$
    iff there is some y such that $R(x, y)$ and $y \Vdash \neg\phi$
    iff $x \Vdash \Diamond\neg\phi$.

Proofs that the other two are valid are similarly routine and left as exercises.

Another important formula which can be seen to be valid is the following:

$$\Box(\phi \to \psi) \land \Box\phi \to \Box\psi.$$

It is sometimes written in the equivalent, but slightly less intuitive, form $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$. This formula scheme is called K in most books about modal logic, honouring the logician S. Kripke who, as we mentioned earlier, invented the so-called 'possible worlds semantics' of Definition 5.4.

To see that K is valid, again suppose we have some world x in some model $\mathcal{M} = (W, R, L)$. We have to show that $x \Vdash \Box(\phi \to \psi) \land \Box\phi \to \Box\psi$. Again referring to Definition 5.4, we assume that $x \Vdash \Box(\phi \to \psi) \land \Box\phi$ and try to prove that $x \Vdash \Box\psi$:

$x \Vdash \Box(\phi \to \psi) \land \Box\phi$
    iff $x \Vdash \Box(\phi \to \psi)$ and $x \Vdash \Box\phi$
    iff for all y with $R(x, y)$, we have $y \Vdash \phi \to \psi$ and $y \Vdash \phi$
        implies that, for all y with $R(x, y)$, we have $y \Vdash \psi$
    iff $x \Vdash \Box\psi$.

There aren't any other interesting valid formulas in basic modal logic. Later, we will see additional valid formulas in extended modal logics of interest.

5.3 Logic engineering

Having looked at the framework for basic modal logic, we turn now to how one may formalise the different modes of truth discussed at the beginning of this chapter. The basic framework is quite general and can be refined in various ways to give us the properties appropriate for the intended applications. Logic engineering is the subject of engineering logics to fit new applications. It is potentially a very broad subject, drawing on all branches of logic, computer science and mathematics. In this chapter, however, we are restricting ourselves to the particular engineering of *modal* logics.

We will consider how to re-engineer basic modal logic to fit the following readings of $\Box\phi$:

- It is necessarily true that ϕ
- It will always be true that ϕ
- It ought to be that ϕ
- Agent Q believes that ϕ
- Agent Q knows that ϕ
- After any execution of program P, ϕ holds.

As modal logic automatically gives us the connective $\Diamond$, which is equivalent to $\neg\Box\neg$, we can find out what the corresponding readings of $\Diamond$ in our system will be. For example, 'it is *not* necessarily true that *not* ϕ' means that it is possibly true that ϕ. You could work this out in steps:

It is *not* necessarily true that ϕ
= it is possible that *not* ϕ.

Therefore,

It is *not* necessarily true that *not* ϕ
= it is possible that *not not* ϕ
= it is possible that ϕ.

Let us work this out with the reading 'agent Q knows ϕ' for $\Box\phi$. Then, $\Diamond\phi$ is read as

agent Q does *not* know *not* ϕ
= as far as Q's knowledge is concerned, ϕ could be the case
= ϕ is consistent with what agent Q knows
= for all agent Q knows, ϕ.

The readings for $\Diamond$ for the other modes are given in Table 5.6.

Table 5.6. The readings of $\Diamond$ corresponding to each reading of $\Box$.

$\Box\phi$	$\Diamond\phi$
It is necessarily true that ϕ	It is possibly true that ϕ
It will always be true that ϕ	Sometime in the future ϕ
It ought to be that ϕ	It is permitted to be that ϕ
Agent Q believes that ϕ	ϕ is consistent with Q's beliefs
Agent Q knows that ϕ	For all Q knows, ϕ
After any execution of program P, ϕ holds	After some execution of P, ϕ holds

5.3.1 The stock of valid formulas

We saw in the last section some valid formulas of basic modal logic, such as instances of the axiom scheme K: $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$ and of the schemes in (5.3). Many other formulas, such as

- $\Box p \to p$
- $\Box p \to \Box\Box p$
- $\neg\Box p \to \Box\neg\Box p$
- $\Diamond\top$

are *not* valid. For example, for each one of these, there is a world in the Kripke model of Figure 5.3 which does not satisfy the formula. The world x_1 satisfies $\Box p$, but it does not satisfy p, so it does not satisfy $\Box p \to p$. If we add $R(x_2, x_1)$ to our model, then x_1 still satisfies $\Box p$ but does not satisfy $\Box\Box p$. Thus, x_1 fails to satisfy $\Box p \to \Box\Box p$. If we change $L(x_4)$ to $\{p, q\}$, then x_4 does not satisfy $\neg\Box p \to \Box\neg\Box p$, because it satisfies $\neg\Box p$, but it does not satisfy $\Box\neg\Box p$ – the path $R(x_4, x_5)R(x_5, x_4)$ serves as a counter example. Finally, x_6 does not satisfy $\Diamond\top$, for this formula states that there is an accessible world satisfying $\top$, which is not the case.

If we are to build a logic capturing the concept of necessity, however, we must surely have that $\Box p \to p$ is valid; for anything which is *necessarily true* is also simply true. Similarly, we would expect $\Box p \to p$ to be valid in the case that $\Box p$ means 'agent Q knows p,' for anything which is known must also be true. We cannot *know* something which is false. We can, however, *believe* falsehoods, so in the case of a logic of belief, we would *not* expect $\Box p \to p$ to be valid.

Part of the job of logic engineering is to determine what formula schemes should be valid and to craft the logic in such a way that precisely those ones are valid.

Table 5.7 shows six interesting readings for $\Box$ and eight formula schemes. For each reading and each formula scheme, we decide whether we should expect the scheme to be valid. Notice that we should only put a tick if the

$\Box\phi$	$\Box\phi \to \phi$	$\Box\phi \to \Box\Box\phi$	$\Diamond\phi \to \Box\Diamond\phi$	$\Diamond\top$	$\Box\phi \to \Diamond\phi$	$\Box\phi \vee \Box\neg\phi$	$\Box(\phi \to \psi) \wedge \Box\phi \to \Box\psi$	$\Diamond\phi \wedge \Diamond\psi \to \Diamond(\phi \wedge \psi)$
It is necessarily true that ϕ	√	√	√	√	√	×	√	×
It will always be true that ϕ	×	√	×	×	×	×	√	×
It ought to be that ϕ	×	×	×	√	√	×	√	×
Agent Q believes that ϕ	×	√	√	√	√	×	√	×
Agent Q knows that ϕ	√	√	√	√	√	×	√	×
After any execut'n of prgrm P, ϕ holds	×	×	×	×	×	×	√	×

Table 5.7. Which formula schemes should hold for these readings of $\Box$?

formula should be valid for all cases of ϕ and ψ. If it could be valid for some cases, but not for others, we put a cross.

There are many points worth noting about Table 5.7. First, observe that it is rather debatable whether to put a tick, or a cross, in some of the cells. We need to be precise about the concept of truth we are trying to formalise, in order to resolve any ambiguity.

Necessity. When we ask ourselves whether $\Box\phi \to \Box\Box\phi$ and $\Diamond\phi \to \Box\Diamond\phi$ should be valid, it seems to depend on what notion of necessity we are referring to. These formulas are valid if that which is necessary is *necessarily* necessary. If we are dealing with *physical necessity*, then this amounts to: are the laws of the universe themselves physically necessary, i.e. do they entail that they should be the laws of the universe? The answer seems to be no. However, if we meant *logical necessity*, it seems that we should give the answer yes, for the laws of logic are meant to be those assertions whose truth cannot be denied. The row is filled on the understanding that we mean logical necessity.

Always in the future. We must be precise about whether or not the future includes the present; this is precisely what the formula $\Box\phi \to \phi$ states. It is a matter of convention whether the future includes the present, or not. In Chapter 3, we saw that CTL adopts the convention that it does. For variety, therefore, let us assume that the future does not include the present in this row of the table. That means that $\Box\phi \to \phi$ fails. What about $\Diamond\top$? It says that there is a future world in which $\top$ is true. In particular, then, there is a future world, i.e. time has no end. Whether we regard this as true or not depends on exactly what notion of 'the future' we are trying to model. We assumed the validity of $\Diamond\top$

in Chapter 3 on CTL since this resulted in an easier presentation of our model-checking algorithms, but we might choose to model it otherwise, as in Table 5.7.

Ought. In this case the formulas $\Box\phi \to \Box\Box\phi$ and $\Diamond\phi \to \Box\Diamond\phi$ state that the moral codes we adopt are themselves forced upon us by morality. This seems not to be the case; for example, we may believe that *'It ought to be the case that we wear a seat-belt,'* but this does not compel us to believe that *'It ought to be the case that we ought to wear a seat-belt.'* However, anything which ought to be so should be permitted to be so; therefore, $\Box\phi \to \Diamond\phi$.

Belief. To decide whether $\Diamond\top$, let us express it as $\neg\Box\bot$, for this is semantically equivalent. It says that agent Q does not believe any contradictions. Here we must be precise about whether we are modelling human beings, with all their foibles and often plainly contradictory beliefs, or whether we are modelling idealised agents that are logically omniscient – i.e. capable of working out the logical consequences of their beliefs. We opt to model the latter concept. The same issue arises when we consider, for example, $\Diamond\phi \to \Box\Diamond\phi$, which – when we rewrite it as $\neg\Box\psi \to \Box\neg\Box\psi$ – says that, if agent Q doesn't believe something, then he believes that he doesn't believe it. Validity of the formula $\Box\phi \vee \Box\neg\phi$ would mean that Q has an opinion on every matter; we suppose this is unlikely. What about $\Diamond\phi \wedge \Diamond\psi \to \Diamond(\phi \wedge \psi)$? Let us rewrite it as $\neg\Diamond(\phi \wedge \psi) \to \neg(\Diamond\phi \wedge \Diamond\psi)$, i.e. $\Box(\neg\phi \vee \neg\psi) \to (\Box\neg\phi \vee \Box\neg\psi)$ or – if we subsume the negations into the ϕ and ψ – the formula $\Box(\phi \vee \psi) \to (\Box\phi \vee \Box\psi)$. This seems not to be valid, for agent Q may be in a situation in which she or he believes that there is a key in the red box, or in the green box, without believing that it is in the red box and also without believing that it is in the green box.

Knowledge. It seems to differ from belief only in respect of the first formula in Table 5.7; while agent Q can have false beliefs, he can only *know* that which is true. In the case of knowledge, the formulas $\Box\phi \to \Box\Box\phi$ and $\neg\Box\psi \to \Box\neg\Box\psi$ are called *positive introspection* and *negative introspection*, respectively, since they state that the agent can introspect upon her knowledge; if she knows something, she knows that she knows it; and if she does not know something, she again knows that she doesn't know it. Clearly, this represents *idealised* knowledge, since most humans – with all their hang-ups and infelicities – do not satisfy these properties. The formula scheme K is sometimes referred to as *logical omniscience* in the logic of knowledge, since it says that the agent's knowledge is closed under logical consequence. This means that the agent knows all the consequences of anything he knows, which is unfortunately (or fortunately?) true only for idealised agents, not humans.

Execution of programs. Not many of our formulas seem to hold in this case. The scheme $\Box\phi \rightarrow \Box\Box\phi$ says that running the program twice is the same as running it once, which is plainly wrong in the case of a program which deducts money from your bank account. The formula $\Diamond\top$ says that there is an execution of the program which terminates; this is false for some programs.

The formula schemes $\Diamond\top$ and $\Box\phi \rightarrow \Diamond\phi$ were seen to be equivalent in the preceding section and, indeed, we see that they get the same pattern of ticks and crosses. We can also show that $\Box\phi \rightarrow \phi$ entails $\Diamond\top$ – i.e. $(\Box\phi \rightarrow \phi) \rightarrow \Diamond\top$ is valid – so whenever the former gets a tick, so should the latter. This is indeed the case, as you can verify in Table 5.7.

5.3.2 Important properties of the accessibility relation

So far, we have been engineering logics at the level of deciding what formulas should be valid for the various readings of $\Box$. We can also engineer logics at the level of Kripke models. For each of our six readings of $\Box$, there is a corresponding reading of the accessibility relation R which will then suggest that R enjoys certain properties such as reflexivity or transitivity.

Let us start with necessity. The clauses

$$\begin{array}{ll} x \Vdash \Box\psi & \text{iff for each } y \in W \text{ with } R(x,y) \text{ we have } y \Vdash \psi \\ x \Vdash \Diamond\psi & \text{iff there is a } y \in W \text{ such that } R(x,y) \text{ and } y \Vdash \psi \end{array}$$

from Definition 5.4 tell us that ϕ is necessarily true at x if ϕ is true in all worlds y accessible from x in a certain way; but accessible in what way? Intuitively, necessarily ϕ is true if ϕ is true in all *possible* worlds; so $R(x,y)$ should be interpreted as meaning that y is a possible world according to the information in x.

In the case of knowledge, we think of $R(x,y)$ as saying: y could be the actual world according to agent Q's knowledge at x. In other words, if the actual world is x, then agent Q – who is not omniscient – cannot rule out the possibility of it being y. If we plug this definition into the clause above for $x \Vdash \Box\phi$, we find that agent Q knows ϕ iff ϕ is true in all the worlds that, for all he knows, could be the actual world. The meaning of R for each of the six readings of $\Box$ is shown in Table 5.8.

Recall that a given binary relation R may be:

- *reflexive*: if, for every $x \in W$, we have $R(x,x)$;
- *symmetric*: if, for every $x, y \in W$, we have $R(x,y)$ implies $R(y,x)$;

Table 5.8. For each reading of $\Box$, the meaning of R is given.

$\Box\phi$	$R(x,y)$
It is necessarily true that ϕ	y is possible world according to the information at x
It will always be true that ϕ	y is a future world of x
It ought to be that ϕ	y is an acceptable world according to the information at x
Agent Q believes that ϕ	y could be the actual world according to Q's beliefs at x
Agent Q knows that ϕ	y could be the actual world according to Q's knowledge at x
After any execution of P, ϕ holds	y is a possible resulting state after execution of P at x

- *serial*: if, for every x there is a y such that $R(x,y)$;
- *transitive*: if, for every $x, y, z \in W$, we have $R(x,y)$ and $R(y,z)$ imply $R(x,z)$;
- *Euclidean*: if, for every $x, y, z \in W$ with $R(x,y)$ and $R(x,z)$, we have $R(y,z)$;
- *functional*: if, for each x there is a unique y such that $R(x,y)$;
- *linear*: if, for every $x, y, z \in W$, we have that $R(x,y)$ and $R(x,z)$ together imply that $R(y,z)$, or y equals z, or $R(z,y)$;
- *total*: if for every $x, y \in W$ we have $R(x,y)$ or $R(y,x)$; and
- an *equivalence relation*: if it is reflexive, symmetric and transitive.

Now, let us consider this question: according to the various readings of R, which of these properties do we expect R to have?

Example 5.9 If $\Box\phi$ means 'agent Q knows ϕ,' then $R(x,y)$ means y could be the actual world according to Q's knowledge at x.

- Should R be reflexive? This would say: x could be the actual world according to Q's knowledge at x. In other words, Q cannot know that things are different from how they really are – i.e., Q cannot have false knowledge. This is a desirable property for R to have. Moreover, it seems to rest on the same intuition – i.e. the impossibility of false knowledge – as the validity of the formula $\Box\phi \to \phi$. Indeed, the validity of this formula and the property of reflexivity are closely related, as we see later on.
- Should R be transitive? It would say: if y is possible according to Q's knowledge at x and z is possible according to her knowledge at y, then z is possible according to her knowledge at x.

  Well, this seems to be true. For suppose it was not true, i.e. at x she knew something preventing z from being the real world. Then, she would know she knew this thing at x; therefore, she would know something at y which prevented z from being the real world; which contradicts our premise.

In this argument, we relied on positive introspection, i.e. the formula $\Box\phi \rightarrow \Box\Box\phi$. Again, we will shortly see that there is a close correspondence between R being transitive and the validity of this formula.

5.3.3 Correspondence theory

We saw in the preceding section that there appeared to be a correspondence between the validity of $\Box\phi \rightarrow \phi$ and the property that the accessibility relation R is reflexive. The connection between them is that both relied on the intuition that anything which is known by an agent is true. Moreover, there also seemed to be a correspondence between $\Box\phi \rightarrow \Box\Box\phi$ and R being transitive; they both seem to assert the property of *positive introspection*, i.e. that which is known is known to be known.

In this section, we will see that there is a precise mathematical relationship between these formulas and properties of R. Indeed, to every formula scheme there corresponds a property of R. From the point of view of logic engineering, it is important to see this relationship, because it helps one to understand the logic being studied. For example, if you believe that a certain formula scheme should be accepted in the system of modal logic you are engineering, then it is well worth looking at the corresponding property of R and checking that this property makes sense for the application, too. Alternatively, the meaning of some formulas may seem difficult to understand, so looking at their corresponding properties of R can help.

To state the relationship between formula schemes and their corresponding properties, we need the notion of a (modal) frame.

Definition 5.10 A frame $\mathcal{F} = (W, R)$ is a set W of worlds and a binary relation R on W.

A frame is like a Kripke model (Definition 5.3), except that it has no labelling function. From any model we can extract a frame, by just forgetting about the labelling function; for example, Figure 5.9 shows the frame extracted from the Kripke model of Figure 5.3. A frame is just a set of worlds and an accessibility relationship between them. It has no information about what atomic formulas are true at the various worlds. However, it is useful to say sometimes that the frame, as a whole, satisfies a formula. This is defined as follows.

Definition 5.11 A frame $\mathcal{F} = (W, R)$ satisfies a formula of basic modal logic ϕ if, for each labelling function $L : W \rightarrow \mathcal{P}(\texttt{Atoms})$ and each $w \in W$,

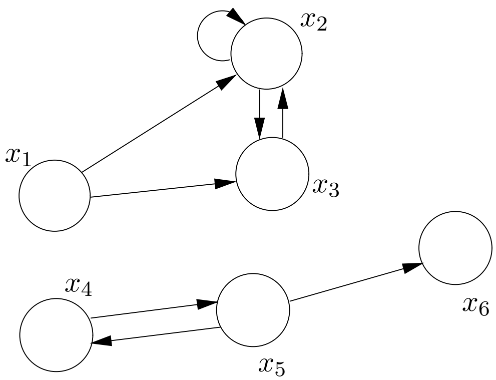


Figure 5.9. The frame of the model in Figure 5.3.

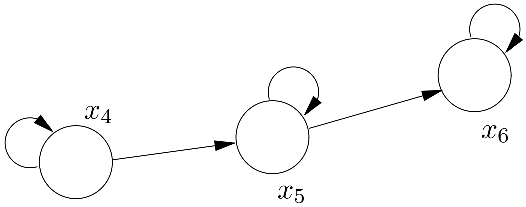


Figure 5.10. Another frame.

the relation $\mathcal{M}, w \Vdash \phi$ holds, where $\mathcal{M} = (W, R, L)$ – recall the definition of $\mathcal{M}, w \Vdash \phi$ on page 310. In that case, we say that $\mathcal{F} \vDash \phi$ holds.

One can show that, if a frame satisfies a formula, then it also satisfies every substitution instance of that formula. Conversely, if a frame satisfies an instance of a formula scheme, it satisfies the whole scheme. This contrasts markedly with models. For example, the model of Figure 5.3 satisfies $p \lor \Diamond p \lor \Diamond\Diamond p$, but doesn't satisfy every instance of $\phi \lor \Diamond\phi \lor \Diamond\Diamond\phi$; for example, x_6 does not satisfy $q \lor \Diamond q \lor \Diamond\Diamond q$. Since frames don't contain any information about the truth or falsity of propositional atoms, they can't distinguish between different atoms; so, if a frame satisfies a formula, it also satisfies the formula scheme obtained by substituting its atoms $p, q, \ldots$ by $\phi, \psi, \ldots$

Examples 5.12 Consider the frame $\mathcal{F}$ in Figure 5.10.

1. $\mathcal{F}$ satisfies the formula $\Box p \to p$. To see this, we have to consider any labelling function of the frame – there are eight such labelling functions, since p could be true or false in each of the three worlds – and show that each world satisfies the formula for each labelling. Rather than really doing this literally, let us

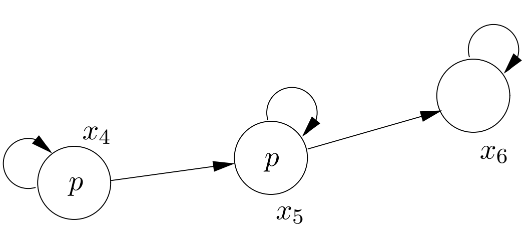


Figure 5.11. A model.

give a generic argument: let x be any world. Suppose that $x \Vdash \Box p$; we want to show $x \Vdash p$. We know that $R(x, x)$ because each x is accessible from itself in the diagram; so, it follows from the clause for $\Box$ in Definition 5.4 that $x \Vdash p$.

2. Therefore, our frame $\mathcal{F}$ satisfies any formula of this shape, i.e. it satisfies the formula scheme $\Box\phi \to \phi$.
3. The frame does not satisfy the formula $\Box p \to \Box\Box p$. For suppose we take the labelling of Figure 5.11; then $x_4 \Vdash \Box p$, but $x_4 \nVdash \Box\Box p$.

If you think about why the frame of Figure 5.10 satisfied $\Box p \to p$ and why it did not satisfy $\Box p \to \Box\Box p$, you will probably guess the following:

Theorem 5.13 Let $\mathcal{F} = (W, R)$ be a frame.

1. The following statements are equivalent:
   - R is reflexive;
   - $\mathcal{F}$ satisfies $\Box\phi \to \phi$;
   - $\mathcal{F}$ satisfies $\Box p \to p$;
2. The following statements are equivalent:
   - R is transitive;
   - $\mathcal{F}$ satisfies $\Box\phi \to \Box\Box\phi$;
   - $\mathcal{F}$ satisfies $\Box p \to \Box\Box p$.

PROOF: Each item 1 and 2 requires us to prove three things: (a) that, if R has the property, then the frame satisfies the formula scheme; and (b) that, if the frame satisfies the formula scheme, it satisfies the instance of it; and (c) that, if the frame satisfies a formula instance, then R has the property.

1. (a) Suppose R is reflexive. Let L be a labelling function, so now $\mathcal{M} = (W, R, L)$ is a model of basic modal logic. We need to show $\mathcal{M} \vDash \Box\phi \to \phi$. That means we need to show $x \Vdash \Box\phi \to \phi$ for any $x \in W$, so pick any x. Use the clause for implication in Definition 5.4. Suppose $x \Vdash \Box\phi$; since $R(x, x)$, it immediately follows from the clause for $\Box$ in Definition 5.4 that $x \Vdash p$. Therefore, we have shown $x \Vdash \Box\phi \to \phi$.
   (b) We just set ϕ to be p.

Table 5.12. Properties of R corresponding to some formulas.

name	formula scheme	property of R
T	$\Box\phi \to \phi$	reflexive
B	$\phi \to \Box\Diamond\phi$	symmetric
D	$\Box\phi \to \Diamond\phi$	serial
4	$\Box\phi \to \Box\Box\phi$	transitive
5	$\Diamond\phi \to \Box\Diamond\phi$	Euclidean
	$\Box\phi \leftrightarrow \Diamond\phi$	functional
	$\Box(\phi \wedge \Box\phi \to \psi) \vee \Box(\psi \wedge \Box\psi \to \phi)$	linear

(c) Suppose the frame satisfies $\Box p \to p$. Take any x; we're going to show $R(x, x)$. Take a labelling function L such that $p \notin L(x)$ and $p \in L(y)$ for all worlds y except x. Proof by contradiction: Assume we don't have $R(x, x)$. Then, $x \Vdash \Box p$, since all the worlds accessible from x satisfy p – this is because all the worlds except x satisfy p; but since $\mathcal{F}$ satisfies $\Box p \to p$, it follows that $x \Vdash \Box p \to p$; therefore, putting $x \Vdash \Box p$ and $x \Vdash \Box p \to p$ together, we get $x \Vdash p$. This is a contradiction to the assumption that we don't have $R(x, x)$, since we said that $p \notin L(x)$. So we must have $R(x, x)$ in our frame!

2. (a) Suppose R is transitive. Let L be a labelling function and $\mathcal{M} = (W, R, L)$. We need to show $M \Vdash \Box\phi \to \Box\Box\phi$. That means we need to show $x \Vdash \Box\phi \to \Box\Box\phi$ for any $x \in W$. Suppose $x \Vdash \Box\phi$; we need to show $x \Vdash \Box\Box\phi$. That is, using the clause for $\Box$ in Definition 5.4, that any y such that $R(x, y)$ satisfies $\Box\phi$; that is, for any y, z with $R(x, y)$ and $R(y, z)$, we have $z \Vdash \phi$.

   Well, suppose we did have y and z with $R(x, y)$ and $R(y, z)$. By the fact that R is transitive, we obtain $R(x, z)$. But we're supposing that $x \Vdash \Box\phi$, so from the meaning of $\Box$ we get $z \Vdash \phi$, which is what we needed to prove.

   (b) Again, just set ϕ to be p.

   (c) Suppose the frame satisfies $\Box p \to \Box\Box p$. Take any x, y and z with $R(x, y)$ and $R(y, z)$; we are going to show $R(x, z)$.

   Define a labelling function L such that $p \notin L(z)$ and $p \in L(w)$ for all worlds w except z. Suppose we don't have $R(x, z)$; then $x \Vdash \Box p$, since $w \Vdash p$ for all $w \neq z$. Using the axiom $\Box p \to \Box\Box p$, it follows that $x \Vdash \Box\Box p$. So $y \Vdash \Box p$ holds since $R(x, y)$. The latter and $R(y, z)$ then render $z \Vdash p$, a contradiction. Thus, we must have $R(x, z)$. □

This picture is completed in Table 5.12, which shows, for a collection of formulas, the corresponding property of R. What this table means mathematically is the following:

Theorem 5.14 A frame $\mathcal{F} = (W, R)$ satisfies a formula scheme in Table 5.12 iff R has the corresponding property in that table.

The names of the formulas in the left-hand column are historical, but have stuck and are still used widely in books.

5.3.4 Some modal logics

The logic engineering approach of this section encourages us to design logics by picking and choosing a set $\mathbb{L}$ of formula schemes, according to the application at hand. Some examples of formula schemes that we may wish to consider for a given application are those in Tables 5.7 and 5.12.

Definition 5.15 Let $\mathbb{L}$ be a set of formula schemes of modal logic and $\Gamma \cup \{\psi\}$ a set of formulas of basic modal logic.

1. The set Γ is closed under substitution instances iff whenever $\phi \in \Gamma$, then any substitution instance of ϕ is also in Γ.
2. Let $\mathbb{L}_c$ be the smallest set containing all instances of $\mathbb{L}$.
3. Γ semantically entails ψ in $\mathbb{L}$ iff $\Gamma \cup \mathbb{L}_c$ semantically entails ψ in basic modal logic. In that case, we say that $\Gamma \vDash_{\mathbb{L}} \psi$ holds.

Thus, we have $\Gamma \vDash_{\mathbb{L}} \psi$ if every Kripke model and every world x satisfying $\Gamma \cup \mathbb{L}_c$ therein also satisfies ψ. Note that for $\mathbb{L} = \emptyset$ this definition is consistent with the one of Definition 5.7, since we then have $\Gamma \cup \mathbb{L}_c = \Gamma$. For logic engineering, we require that $\mathbb{L}$ be

- closed under substitution instances; otherwise, we won't be able to characterize $\mathbb{L}_c$ in terms of properties of the accessibility relation; and
- *consistent* in that there is a frame $\mathcal{F}$ such that $\mathcal{F} \vDash \phi$ holds for all $\phi \in \mathbb{L}$; otherwise, $\Gamma \vDash_{\mathbb{L}} \psi$ holds for all Γ and ψ! In most applications of logic engineering, consistency is easy to establish.

We now study a few important modal logics that extend basic modal logic with a consistent set of formula schemes $\mathbb{L}$.

The modal logic K The weakest modal logic doesn't have any chosen formula schemes, like those of Tables 5.7 and 5.12. So $\mathbb{L} = \emptyset$ and this modal logic is called K as it satisfies all instances of the formula scheme K; modal logics with this property are called normal and all modal logics we study in this text are normal.

The modal logic KT45 A well-known modal logic is KT45 – also called S5 in the technical literature – where $\mathbb{L} = \{T, 4, 5\}$ with T, 4 and 5 from Table 5.12. This logic is used to reason about knowledge; $\Box\phi$ means that the agent Q knows ϕ. Table 5.12 tell us, respectively, that

T. Truth: the agent Q knows only true things.

4. Positive introspection: if the agent Q knows something, then she knows that she knows it.
5. Negative introspection: if the agent Q doesn't know something, then she knows that she doesn't know it.

In this application, the formula scheme K means logical omniscience: the agent's knowledge is closed under logical consequence. Note that these properties represent idealisations of knowledge. Human knowledge has none of these properties! Even computer agents may not have them all. There are several attempts in the literature to define logics of knowledge that are more realistic, but we will not consider them here.

The semantics of the logic KT45 must consider only relations R which are: reflexive (T), transitive (4) and Euclidean (5).

Fact 5.16 A relation is reflexive, transitive and Euclidean iff it is reflexive, transitive and symmetric, i.e. if it is an equivalence relation.

KT45 is simpler than K in the sense that it has few essentially different ways of composing modalities.

Theorem 5.17 Any sequence of modal operators and negations in KT45 is equivalent to one of the following: $-$, $\Box$, $\Diamond$, $\neg$, $\neg\Box$ and $\neg\Diamond$, where $-$ indicates the absence of any negation or modality.

The modal logic KT4 The modal logic KT4, that is $\mathbb{L}$ equals $\{T, 4\}$, is also called S4 in the literature. Correspondence theory tells us that its models are precisely the Kripke models $\mathcal{M} = (W, R, L)$, where R is reflexive and transitive. Such structures are often very useful in computer science. For example, if ϕ stands for the type of a piece of code – ϕ could be $\texttt{int} \times \texttt{int} \rightarrow \texttt{bool}$, indicating some code which expects a pair of integers as input and outputs a boolean value – then $\Box\phi$ could stand for *residual code* of type ϕ. Thus, in the current world x this code would not have to be executed, but could be saved (= residualised) for execution at a later computation stage. The formula scheme $\Box\phi \rightarrow \phi$, the axiom T, then means that code may be executed right away, whereas the formula scheme $\Box\phi \rightarrow \Box\Box\phi$, the axiom 4, allows that residual code remain residual, i.e. we can repeatedly postpone its execution in future computation stages. Such type systems have important applications in the specialisation and partial evaluation of code. We refer the interested reader to the bibliographic notes at the end of the chapter.

Theorem 5.18 Any sequence of modal operators and negations in KT4 is equivalent to one of the following: $-$, $\Box$, $\Diamond$, $\Box\Diamond$, $\Diamond\Box$, $\Box\Diamond\Box$, $\Diamond\Box\Diamond$, $\neg$, $\neg\Box$, $\neg\Diamond$, $\neg\Box\Diamond$, $\neg\Diamond\Box$, $\neg\Box\Diamond\Box$ and $\neg\Diamond\Box\Diamond$.

Intuitionistic propositional logic In Chapter 1, we gave a natural deduction system for propositional logic which was sound and complete with respect to semantic entailment based on truth tables. We also pointed out that the proof rules PBC, LEM and $\neg\neg\mathrm{e}$ are questionable in certain computational situations. If we disallow their usage in natural deduction proofs, we obtain a logic, called *intuitionistic propositional logic*, together with its own proof theory. So far so good; but it is less clear what sort of semantics one could have for such a logic – again with soundness and completeness in mind. This is where certain models of KT4 will do the job quite nicely. Recall that correspondence theory implies that a model $\mathcal{M} = (W, R, L)$ of KT4 is such that R is reflexive and transitive. The only additional requirement we impose on a model for intuitionistic propositional logic is that its labelling function L be *monotone* in R: $R(x, y)$ implies that $L(x)$ is a subset of $L(y)$. This models that the truth of atomic positive formulas persist throughout the worlds that are reachable from a given world.

Definition 5.19 A model of intuitionistic propositional logic is a model $\mathcal{M} = (W, R, L)$ of KT4 such that $R(x, y)$ always implies $L(x) \subseteq L(y)$. Given a propositional logic formula as in (1.3), we define $x \Vdash \phi$ as in Definition 5.4 exception for the clauses $\rightarrow$ and $\neg$. For $\phi_1 \rightarrow \phi_2$ we define $x \Vdash \phi_1 \rightarrow \phi_2$ iff for all y with $R(x, y)$ we have $y \Vdash \phi_2$ whenever we have $y \Vdash \phi_1$. For $\neg\phi$ we define $x \Vdash \neg\phi$ iff for all y with $R(x, y)$ we have $y \nVdash \phi$.

As an example, consider the model $W = \{x, y\}$ with accessibility relation $R = \{(x, x), (x, y), (y, y)\}$, which is indeed reflexive and transitive. For a labelling function L with $L(x) = \emptyset$ and $L(y) = \{p\}$, we claim that $x \nVdash p \vee \neg p$. (Recall that $p \vee \neg p$ is an instance of LEM which we proved in Chapter 1 with the full natural deduction calculus.) We do not have $x \Vdash p$, for p is not in the set $L(x)$ which is empty. Thus, Definition 5.4 for the case $\vee$ implies that $x \Vdash p \vee \neg p$ can hold only if $x \Vdash \neg p$ holds. But $x \Vdash \neg p$ simply does not hold, since there is a world y with $R(x, y)$ such that $y \Vdash p$ holds, for $p \in L(y)$. The availability of possible worlds in the models of KT4 together with a 'modal interpretation' of $\rightarrow$ and $\neg$ breaks down the validity of the theorem LEM in classical logic.

One can now define semantic entailment in the same manner as for modal logics. Then, one can prove soundness and completeness of the reduced natural deduction system with respect to this semantic entailment, but those proofs are beyond the scope of this book.

5.4 Natural deduction

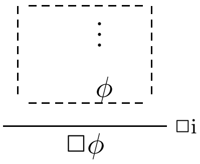
Verifying semantic entailment $\Gamma \vDash_{\mathbb{L}} \psi$ by appealing to its definition directly would be rather difficult. We would have to consider every Kripke model that satisfies all formulas of Γ and every world in it. Fortunately, we have a much more usable approach, which is an extension, respectively adaptation, of the systems of natural deduction met in Chapters 1 and 2. Recall that we presented natural deduction proofs as linear representations of proof trees which may involve proof boxes which control the scope of assumptions, or quantifiers. The proof boxes have formulas and/or other boxes inside them. There are rules which dictate how to construct proofs. Boxes open with an *assumption*; when a box is closed – in accordance with a rule – we say that its assumption is *discharged.* Formulas may be repeated and brought into boxes, but may not be brought out of boxes. Every formula must have some justification to its right: a justification can be the name of a rule, or the word 'assumption,' or an instance of the proof rule copy; see e.g. page 13.

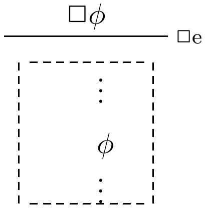
Natural deduction works in a very similar way for modal logic. The main difference is that we introduce a new kind of proof box, to be drawn with dashed lines. This is required for the rules for the connective $\Box$. The dashed proof box has a completely different role from the solid one. As we saw in Chapter 1, going into a solid proof box means making an assumption. Going into a dashed box means *reasoning in an arbitrary accessible world.* If at any point in a proof we have $\Box\phi$, we could open a dashed box and put ϕ in it. Then, we could work on this ϕ, to obtain, for example, ψ. Now we could come out of the dashed box and, since we have shown ψ in an arbitrary accessible world, we may deduce $\Box\psi$ in the world outside the dashed box.

Thus, the rules for bringing formulas into dashed boxes and taking formulas out of them are the following:

- Wherever $\Box\phi$ occurs in a proof, ϕ may be put into a subsequent dashed box.
- Wherever ψ occurs at the end of a dashed box, $\Box\psi$ may be put after that dashed box.

We have thus added two rules, $\Box$ introduction and $\Box$ elimination:





In modal logic, natural deduction proofs contain both solid and dashed boxes, nested in any way. Note that there are no explicit rules for $\Diamond$, which must be written $\neg\Box\neg$ in proofs.

The extra rules for KT45 The rules $\Box$i and $\Box$e are sufficient for capturing semantic entailment of the modal logic K. Stronger modal logics, e.g. KT45, require extra rules if one wants to capture their semantic entailment via proofs. In the case of KT45, this extra strength is expressed by rule schemes for the axioms T, 4 and 5:

$$\frac{\Box\phi}{\phi}\,T \qquad\qquad \frac{\Box\phi}{\Box\Box\phi}\,4 \qquad\qquad \frac{\neg\Box\phi}{\Box\neg\Box\phi}\,5$$

An equivalent alternative to the rules 4 and 5 would be to stipulate relaxations of the rules about moving formulas in and out of dashed boxes. Since rule 4 allows us to double-up boxes, we could instead think of it as allowing us to move formulas beginning with $\Box$ into dashed boxes. Similarly, axiom 5 has the effect of allowing us to move formulas beginning with $\neg\Box$ into dashed boxes. Since 5 is a scheme and since ϕ and $\neg\neg\phi$ are equivalent in basic modal logic, we could write $\neg\phi$ instead of ϕ throughout without changing the expressive power and meaning of that axiom.

Definition 5.20 Let $\mathbb{L}$ be a set of formula schemes. We say that $\Gamma \vdash_{\mathbb{L}} \psi$ is valid if ψ has a proof in the natural deduction system for basic modal logic extended with the axioms from $\mathbb{L}$ and premises from Γ.

Examples 5.21 We show that the following sequents are valid:

1. $\vdash_{\mathrm{K}} \Box p \wedge \Box q \to \Box(p \wedge q)$.

1	$\Box p \wedge \Box q$	assumption
2	$\Box p$	$\wedge$e$_1$ 1
3	$\Box q$	$\wedge$e$_2$ 1
4	p	$\Box$e 2
5	q	$\Box$e 3
6	$p \wedge q$	$\wedge$i 4, 5
7	$\Box(p \wedge q)$	$\Box$i 4−6
8	$\Box p \wedge \Box q \to \Box(p \wedge q)$	$\to$i 1−7

2. $\vdash_{\mathrm{KT45}} p \to \Box\Diamond p$.

1	p	assumption
2	$\Box\neg p$	assumption
3	$\neg p$	T 2
4	$\bot$	$\neg$e 1, 3
5	$\neg\Box\neg p$	$\neg$i 2–4
6	$\Box\neg\Box\neg p$	axiom 5 on line 5
7	$p \to \Box\neg\Box\neg p$	$\to$i 1–6

3. $\vdash_{\mathrm{KT45}} \Box\Diamond\Box p \to \Box p$.

1	$\Box\neg\Box\neg\Box p$	assumption
2	$\neg\Box\neg\Box p$	$\Box$e 1
3	$\neg\Box p$	assumption
4	$\Box\neg\Box p$	axiom 5 on line 3
5	$\bot$	$\neg$e 4, 2
6	$\neg\neg\Box p$	$\neg$i 3–5
7	$\Box p$	$\neg\neg$e 6
8	p	T 7
9	$\Box p$	$\Box$i 2–8
10	$\Box\neg\Box\neg\Box p \to \Box p$	$\to$i 1–9

5.5 Reasoning about knowledge in a multi-agent system

In a *multi-agent system*, different agents have different knowledge of the world. An agent may need to reason about its own knowledge about the world; it may also need to reason about what other agents know about the world. For example, in a bargaining situation, the seller of a car must consider what a buyer knows about the car's value. The buyer must also consider what the seller knows about what the buyer knows about that value and so on.

Reasoning about knowledge refers to the idea that agents in a group take into account not only the facts of the world, but also the knowledge of other agents in the group. Applications of this idea include: games, economics, cryptography and protocols. It is not very easy for humans to follow the thread of such nested sentences as

> *Dean doesn't know whether Nixon knows that Dean knows that Nixon knows that McCord burgled O'Brien's office at Watergate.*

However, computer agents are better than humans in this respect.

5.5.1 Some examples

We start with some classic examples about reasoning in a multi-agent environment. Then, in the next section, we engineer a modal logic which allows for a formal representation of these examples via sequents and which solves them by proving them in a natural deduction system.

The wise-men puzzle There are three wise men. It's common knowledge – known by everyone and known to be known by everyone, etc. – that there are three red hats and two white hats. The king puts a hat on each of the wise men in such a way that they are not able to see their own hat, and asks each one in turn whether they know the colour of the hat on their head. Suppose the first man says he does not know; then the second says he does not know either.

It follows that the third man must be able to say that he knows the colour of his hat. Why is this? What colour has the third man's hat?

To answer these questions, let us enumerate the seven possibilities which exist: they are

R R R
R R W
R W R
R W W

W R R
W R W
W W R

where, for example, R W W refers to the situation that the first, second and third men have red, white and white hats, respectively. The eighth possibility, W W W, is ruled out as there are only two white hats.

Now let's think of it from the second and third men's point of view. When they hear the first man speak, they can rule out the possibility of the true situation being R W W, because if it were this situation, then the first man, seeing that the others were wearing white hats and knowing that there are only two white hats, would have concluded that his hat must be red. As he said that he did not know, the true situation cannot be R W W. Notice that the second and third men must be intelligent in order to perform this reasoning; and they must know that the first man is intelligent and truthful as well. In the puzzle, we assume the truthfulness and intelligence and perceptiveness of the men are common knowledge – known by everyone and known to be known by everyone, etc.

When the third man hears the second man speak, he can rule out the possibility of the true situation being W R W, for similar reasons: if it were that, the second man would have said that he knew his hat was red, but he did not say this. Moreover, the third man can also rule out the situation R R W when he hears the second man's answer, for this reason: if the second man had seen that the first was wearing red and the third white, he would have known that it must be R W W or R R W; but he would have known from the first man's answer that it couldn't be R W W, so he would have concluded it was R R W and that he was wearing a red hat; but he did not draw this conclusion, so, reasons the third man, it cannot be R R W.

Having heard the first and second men speak, the third man has eliminated R W W, W R W and R R W; leaving only R R R, R W R, W R R and W W R. In all of these he is wearing a red hat, so he concludes that he must be wearing a red hat.

Notice that the men learn a lot from hearing the other men speak. We emphasise again the importance of the assumption that they tell the truth about their state of knowledge and are perceptive and intelligent enough to come to correct conclusions. Indeed, it is not enough that the three men are truthful, perceptive and intelligent; they must be known to be so by the others and, in later examples, this fact must also be known etc. Therefore, we assume that all this is common knowledge.

The muddy-children puzzle This is one of the many variations on the wise-men puzzle; a difference is that the questions are asked in parallel rather than sequentially. There is a large group of children playing in the garden – their perceptiveness, truthfulness and intelligence being common knowledge, it goes without saying. A certain number of children, say $k \geq 1$, get mud on their foreheads. Each child can see the mud on others, but not on his own forehead. If $k > 1$, then each child can see another with mud on its forehead, so each one knows that at least one in the group is muddy. Consider these two scenarios:

Scenario 1. The father repeatedly asks the question 'Does any of you know whether you have mud on your own forehead?' The first time they all answer 'no;' but, unlike in the wise-men example, they don't learn anything by hearing the others answer 'no,' so they go on answering 'no' to the father's repeated questions.

Scenario 2. The father first announces that at least one of them is muddy – which is something they know already; and then, as before, he repeatedly asks them 'Does any of you know whether you have mud on your own forehead?' The first time they all answer 'no.' Indeed, they go on answering 'no' to the first $k - 1$ repetitions of that same question; but at the kth those with muddy foreheads are able to answer 'yes.'

At first sight, it seems rather puzzling that the two scenarios are different, given that the only difference in the events leading up to them is that in the second one the father announces something that they already know. It would be wrong, however, to conclude that the children learn nothing from this announcement. Although everyone knows the content of the announcement, the father's saying it makes it common knowledge among them, so now they all know that everyone else knows it, etc. This is the crucial difference between the two scenarios.

To understand scenario 2, consider a few cases of k.

$k = 1$, i.e. just one child has mud. That child is immediately able to answer 'yes,' since she has heard the father and doesn't see any other child with mud.

$k = 2$, say only the children Ramon and Candy have mud. Everyone answers 'no' the first time. Now Ramon thinks: since Candy answered 'no' the first time, she must see someone with mud. Well, the only person I can see with mud is Candy, so if she can see someone else it must be me. So Ramon answers 'yes' the second time. Candy reasons symmetrically about Ramon and also answers 'yes' the second time round.

$k = 3$, say only the children Alice, Bob, and Charlie have mud. Everyone answers 'no' the first two times. But now Alice thinks: if it was just Bob and Charlie with mud, they would have answered 'yes' the second time; making the argument for $k = 2$ above. So there must be a third person with mud; since I can see only Bob and Charlie having mud, the third person must be me. So Alice answers 'yes' the third time. For symmetrical reasons, so do Bob and Charlie.

And similarly for other cases of k.

To see that it was not common knowledge before the father's announcement that one of the children was muddy, consider again $k = 2$, with Ramon and Candy. Of course, Ramon and Candy both know someone is muddy – they see each other; but, for example, Ramon doesn't know that Candy knows that someone is dirty. For all Ramon knows, Candy might be the only dirty one and therefore not be able to see a dirty child.

5.5.2 The modal logic KT45^n

We now generalise the modal logic KT45 given in Section 5.3.4. Instead of having just one $\Box$, it will have many, one for each agent i from a fixed set $\mathcal{A} = \{1, 2, \ldots, n\}$ of agents. We write those modal connectives as K_i (for each agent $i \in \mathcal{A}$); the K is to emphasise the application to *knowledge*. We assume a collection $p, q, r, \ldots$ of atomic formulas. The formula $K_i\, p$ means that agent i knows p; so, for example, $K_1\, p \wedge K_1 \neg K_2 K_1\, p$ means that agent 1 knows p, but knows that agent 2 doesn't know he knows it.

We also have the modal connectives E_G, where G is any subset of $\mathcal{A}$. The formula $E_G\, p$ means everyone in the group G knows p. If $G = \{1, 2, 3, \ldots, n\}$, then $E_G\, p$ is equivalent to $K_1\, p \wedge K_2\, p \wedge \cdots \wedge K_n\, p$. We assume similar binding priorities to those put forward on page 307.

Convention 5.22 The binding priorities of KT45^n are the ones of basic modal logic, if we think of each modality K_i, E_G and C_G as 'being' $\Box$.

One might think that ϕ could not be more widely known than everyone knowing it, but this is not the case. It could be, for example, that everyone knows ϕ, but they might not know that they all know it. If ϕ is supposed to be a secret, it might be that you and your friend both know it, but your friend does not know that you know it and you don't know that your friend knows it. Thus, $E_G E_G\, \phi$ is a state of knowledge even greater than $E_G\, \phi$ and $E_G E_G E_G\, \phi$ is greater still. We say that ϕ is *common knowledge among* G, written $C_G\, \phi$, if everyone knows ϕ and everyone knows that everyone knows it; and everyone knows that; and knows *that* etc. So we may think of $C_G\, \phi$ as an infinite conjunction

$$E_G\, \phi \wedge E_G E_G\, \phi \wedge E_G E_G E_G\, \phi \wedge \ldots .$$

However, since our logics only have finite conjunctions, we cannot reduce C_G to something which is already in the logic. We have to express the *infinite* aspect of C_G via its semantics and retain it as an additional modal connective. Finally, $D_G\, \phi$ means the knowledge of ϕ is distributed among the group G; although no-one in G may know it, they would be able to work it out if they put their heads together and combined the information distributed among them.

Definition 5.23 A formula ϕ in the multi-modal logic of KT45^n is defined by the following grammar:

$$\begin{aligned}\phi ::= {} & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi) \mid \\ & (K_i\, \phi) \mid (E_G\, \phi) \mid (C_G\, \phi) \mid (D_G\, \phi)\end{aligned}$$

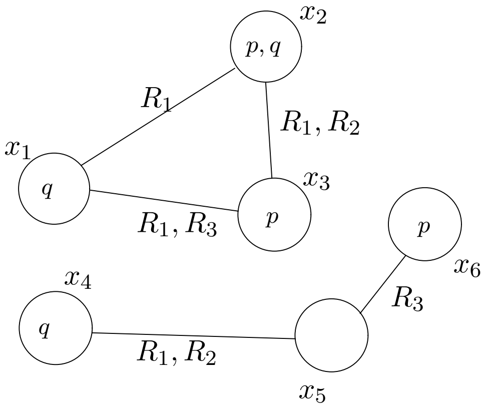


Figure 5.13. A KT45^n model for $n = 3$.

where p is any atomic formula, $i \in \mathcal{A}$ and $G \subseteq \mathcal{A}$. We simply write E, C and D without subscripts if we refer to $E_{\mathcal{A}}$, $C_{\mathcal{A}}$ and $D_{\mathcal{A}}$.

Compare this definition with Definition 5.1. Instead of $\Box$, we have several modalities K_i and we also have E_G, C_G and D_G for each $G \subseteq \mathcal{A}$. Actually, all of these connectives will shortly be seen to be 'box-like' rather than 'diamond-like', in the sense that they distribute over $\wedge$ rather than over $\vee$ – compare this to the discussion of equivalences on page 308. The 'diamond-like' correspondents of these connectives are not explicitly in the language, but may of course be obtained using negations, i.e. $\neg K_i \neg$, $\neg C_G \neg$ etc.

Definition 5.24 A model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ of the multi-modal logic KT45^n with the set $\mathcal{A}$ of n agents is specified by three things:

1. a set W of possible worlds;
2. for each $i \in \mathcal{A}$, an equivalence relation R_i on W ($R_i \subseteq W \times W$), called the accessibility relations; and
3. a labelling function $L : W \to \mathcal{P}(\texttt{Atoms})$.

Compare this with Definition 5.3. The difference is that, instead of just one accessibility relation, we now have a family, one for each agent in $\mathcal{A}$; and we assume the accessibility relations are equivalence relations.

We exploit these properties of R_i in the graphical illustrations of Kripke models for KT45^n. For example, a model of KT45^3 with set of worlds $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ is shown in Figure 5.13. The links between the worlds have to be labelled with the name of the accessibility relation, since we have several relations. For example, x_1 and x_2 are related by R_1, whereas x_4 and

x_5 are related both by R_1 and by R_2. We simplify by no longer requiring arrows on the links. This is because we know that the relations are symmetric, so the links are bi-directional. Moreover, the relations are also reflexive, so there should be loops like the one on x_4 in Figure 5.11 in all the worlds and for all of the relations. We can simply omit these from the diagram, since we don't need to distinguish between worlds which are self-related and those which are not.

Definition 5.25 Take a model $\mathcal{M} = (W, (R_i)_{i\in\mathcal{A}}, L)$ of KT45^n and a world $x \in W$. We define when ϕ is true in x via a satisfaction relation $x \Vdash \phi$ by induction on ϕ:

$$\begin{array}{rl}
x \Vdash p & \text{iff } p \in L(x) \\
x \Vdash \neg\phi & \text{iff } x \nVdash \phi \\
x \Vdash \phi \wedge \psi & \text{iff } x \Vdash \phi \text{ and } x \Vdash \psi \\
x \Vdash \phi \vee \psi & \text{iff } x \Vdash \phi \text{ or } x \Vdash \psi \\
x \Vdash \phi \to \psi & \text{iff } x \Vdash \psi \text{ whenever we have } x \Vdash \phi \\
x \Vdash K_i\,\psi & \text{iff, for each } y \in W,\ R_i(x,y) \text{ implies } y \Vdash \psi \\
x \Vdash E_G\,\psi & \text{iff, for each } i \in G,\ x \Vdash K_i\,\psi \\
x \Vdash C_G\,\psi & \text{iff, for each } k \geq 1, \text{ we have } x \Vdash E_G^k\psi, \\
 & \quad \text{where } E_G^k \text{ means } E_G E_G \dots E_G - k \text{ times} \\
x \Vdash D_G\,\psi & \text{iff, for each } y \in W, \text{ we have } y \Vdash \psi, \\
 & \quad \text{whenever } R_i(x,y) \text{ for all } i \in G.
\end{array}$$

Again, we write $\mathcal{M}, x \Vdash \phi$ if we want to emphasise the model $\mathcal{M}$.

Compare this with Definition 5.4. The cases for the boolean connectives are the same as for basic modal logic. Each K_i behaves like a $\Box$, but refers to its own accessibility relation R_i. As already stated, there are no equivalents of $\Diamond$, but we can recover them as $\neg K_i \neg$. The connective E_G is defined in terms of the K_i and C_G is defined in terms of E_G.

Many of the results we had for basic modal logic with a single accessibility relation also hold in this more general setting of several accessibility relations. Summarising,

- a *frame* $\mathcal{F}$ for KT45^n $(W, (R_i)_{i\in\mathcal{A}})$ for the modal logic KT45^n is a set W of worlds and, for each $i \in \mathcal{A}$, an equivalence relation R_i on W.
- a frame $\mathcal{F} = (W, (R_i)_{i\in\mathcal{A}})$ for KT45^n is said to satisfy ϕ if, for each labelling function $L\colon W \to \mathcal{P}(\texttt{Atoms})$ and each $w \in W$, we have $\mathcal{M}, w \Vdash \phi$ holds, where $\mathcal{M} = (W, (R_i)_{i\in\mathcal{A}}, L)$. In that case, we say that $\mathcal{F} \vDash \phi$ holds.

The following theorem is useful for answering questions about formulas involving E and C. Let $\mathcal{M} = (W,\ (R_i)_{i\in\mathcal{A}}, L)$ be a model for KT45^n

and $x, y \in W$. We say that y is *G-reachable in k steps* from x if there are $w_1, w_2, \ldots, w_{k-1} \in W$ and $i_1, i_2, \ldots, i_k$ in G such that

$$x\, R_{i_1}\, w_1\, R_{i_2}\, w_2\, \ldots\, R_{i_{k-1}}\, w_{k-1}\, R_{i_k}\, y$$

meaning $R_{i_1}(x, w_1)$, $R_{i_2}(w_1, w_2)$, $\ldots$, $R_{i_k}(w_k, y)$. We also say that y is *G-reachable* from x if there is some k such that it is G-reachable in k steps.

Theorem 5.26

1. $x \Vdash E_G^k \phi$ iff, for all y that are G-reachable from x in k steps, we have $y \Vdash \phi$.
2. $x \Vdash C_G\, \phi$ iff, for all y that are G-reachable from x, we have $y \Vdash \phi$.

PROOF:

1. First, suppose $y \Vdash \phi$ for all y G-reachable from x in k steps. We will prove that $x \Vdash E_G^k \phi$ holds. It is sufficient to show that $x \Vdash K_{i_1} K_{i_2} \ldots K_{i_k}\, \phi$ for any $i_1, i_2, \ldots, i_k \in G$. Take any $i_1, i_2, \ldots, i_k \in G$ and any w_1, $w_2, \ldots$, w_{k-1} and y such that there is a path of the form $x\, R_{i_1}\, w_1\, R_{i_2}\, w_2\, \ldots\, R_{i_{k-1}}\, w_{k-1}\, R_{i_k}\, y$. Since y is G-reachable from x in k steps, we have $y \Vdash \phi$ by our assumption, so $x \Vdash K_{i_1} K_{i_2} \ldots K_{i_k}\, \phi$ as required.
   Conversely, suppose $x \Vdash E_G^k \phi$ holds and y is G-reachable from x in k steps. We must show that $y \Vdash \phi$ holds. Take $i_1, i_2, \ldots, i_k$ by G-reachability; since $x \Vdash E_G^k \phi$ implies $x \Vdash K_{i_1} K_{i_2} \ldots K_{i_k}\, \phi$, we have $y \Vdash \phi$.
2. This argument is similar.

Some valid formulas in KT45^n The formula K holds for the connectives K_i, E_G, C_G and D_G, i.e. we have the corresponding formula schemes

$$\begin{gathered}
K_i\, \phi \wedge K_i\, (\phi \to \psi) \to K_i\, \psi \\
E_G\, \phi \wedge E_G\, (\phi \to \psi) \to E_G\, \psi \\
C_G\, \phi \wedge C_G\, (\phi \to \psi) \to C_G\, \psi \\
D_G\, \phi \wedge D_G\, (\phi \to \psi) \to D_G\, \psi.
\end{gathered}$$

This means that these different 'levels' of knowledge are closed under logical consequence. For example, if certain facts are common knowledge and some other fact follows logically from them, then that fact is also common knowledge.

Observe that E, C and D are 'box-like' connectives, in the sense that they quantify universally over certain accessibility relations. That is to say, we may define the relations R_{E_G}, R_{D_G} and R_{C_G} in terms of the relations R_i, as follows:

$$\begin{array}{llll}
R_{E_G}(x, y) & \text{iff} & R_i(x, y) & \text{for some } i \in G \\
R_{D_G}(x, y) & \text{iff} & R_i(x, y) & \text{for all } i \in G \\
R_{C_G}(x, y) & \text{iff} & R_{E_G}^k(x, y) & \text{for each } k \geq 1.
\end{array}$$

It follows from this that E_G, D_G and C_G satisfy the K formula with respect to the accessibility relations R_{E_G}, R_{D_G} and R_{C_G}, respectively.

What about other valid formulas? Since we have stipulated that the relations R_i are equivalence relations, it follows from the multi-modal analogues of Theorem 5.13 and Table 5.12 that the following formulas are valid in KT45^n for each agent i:

$K_i\,\phi \to K_i K_i\,\phi$	positive introspection
$\neg K_i\,\phi \to K_i \neg K_i\,\phi$	negative introspection
$K_i\,\phi \to \phi$	truth.

These formulas also hold for D_G, since R_{D_G} is also an equivalence relation, but these don't automatically generalise for E_G and C_G. For example, $E_G\,\phi \to E_G E_G\,\phi$ is not valid; if it were valid, it would imply that common knowledge was nothing more than knowledge by everybody. The scheme $\neg E_G\,\phi \to E_G \neg E_G\,\phi$ is also not valid. The failure of these formulas to be valid can be traced to the fact that R_{E_G} is not necessarily an equivalence relation, even though each R_i is an equivalence relation. However, R_{E_G} is reflexive, so $E_G\,\phi \to \phi$ is valid, provided that $G \neq \emptyset$. If $G = \emptyset$, then $E_G\,\phi$ holds vacuously, even if ϕ is false.

Since R_{C_G} is an equivalence relation, the formulas T, 4 and 5 above do hold for C_G, although the third one still requires the condition that $G \neq \emptyset$.

5.5.3 Natural deduction for KT45^n

The proof system for KT45 is easily extended to KT45^n; but for simplicity, we omit reference to the connective D.

1. The dashed boxes now come in different 'flavours' for different modal connectives; we'll indicate the modality in the top left corner of the dashed box.
2. The axioms T, 4 and 5 can be used for any K_i, whereas axioms 4 and 5 can be used for C_G, but not for E_G – recall the discussion in Section 5.5.2.
3. In the rule CE, we may deduce $E_G^k\phi$ from $C_G\,\phi$ for any k; or we could go directly to $K_{i_1} \ldots K_{i_k}\,\phi$ for any agents $i_1, \ldots, i_{k\in G}$ by using the rule CK. Strictly speaking, these rules are a whole set of such rules, one for each choice of k and $i_1, \ldots, i_k$, but we refer to all of them as CE and CK respectively.
4. Applying rule EK_i, we may deduce $K_i\,\phi$ from $E_G\,\phi$ for any $i \in G$. From $\bigwedge_{i\in G} K_i\,\phi$ we may deduce $E_G\,\phi$ by virtue of rule KE. Note that the proof rule EK_i is like a generalised and-elimination rule, whereas KE behaves like an and-introduction rule.

The proof rules for KT45^n are summarised in Figure 5.14. As before, we can think of the rules $K4$ and $K5$ and $C4$ and $C5$ as relaxations of the

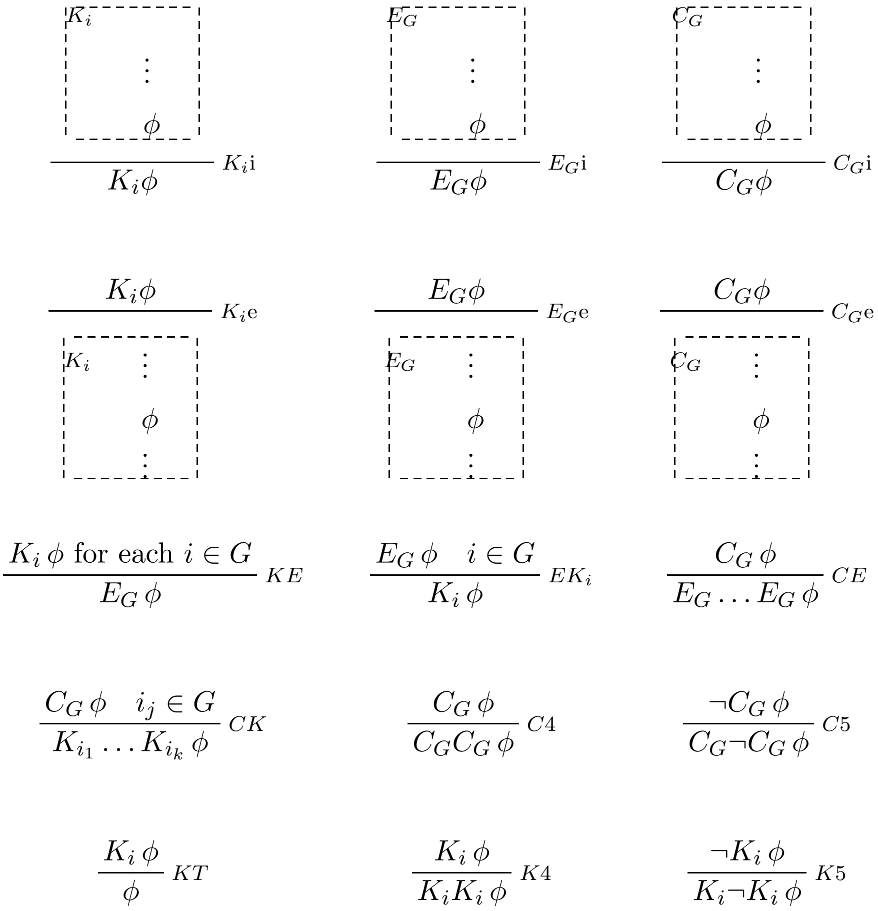


Figure 5.14. Natural deduction rules for KT45^n.

rules about moving formulas in and out of dashed proof boxes. Since rule $K4$ allows us to double-up K_i, we could instead think of it as allowing us to move formulas beginning with K_i into K_i-dashed boxes. Similarly, rule $C5$ has the effect of allowing us to move formulas beginning with $\neg C_G$ into C_G-dashed boxes.

An intuitive way of thinking about the dashed boxes is that formulas in them are known to the agent in question. When you open a K_i-dashed box, you are considering what agent i knows. It's quite intuitive that an ordinary formula ϕ cannot be brought into such a dashed box, because the mere truth of ϕ does not mean that agent i knows it. In particular, you can't use the rule $\neg$i if one of the premises of the rule is outside the dashed box you're working in.

1		$C(p \vee q)$	premise			
2		$K_1(K_2\, p \vee K_2\, \neg p)$	premise			
3		$K_1 \neg K_2\, q$	premise			
4		$K_1 K_2\, (p \vee q)$	CK 1			
5	K_1	$K_2\, (p \vee q)$	K_1e 4			
6		$K_2\, p \vee K_2\, \neg p$	K_1e 2			
7		$\neg K_2\, q$	K_1e 3			
8		$K_2\, p$	assumption		$K_2\, \neg p$	assumption
9		p	axiom T 8	K_2	$\neg p$	K_2e 8
10					$p \vee q$	K_2e 5
11					q	prop 9, 10
12					$K_2\, q$	K_2i 9–11
13					$\bot$	$\neg$e 12, 7
14					p	$\bot$e 13
15		p	$\vee$e 6, 8–14, 8–14			
16		$K_1\, p$	K_1i 5–15			

Figure 5.15. A proof of $C(p \vee q)$, $K_1(K_2 p \vee K_2\, \neg p)$, $K_1 \neg K_2\, q \vdash K_1\, p$.

Observe the power of $C\, \phi$ in the premises: we can bring ϕ into *any* dashed box by the application of the rules CK and K_ie, no matter how deeply nested boxes are. The rule $E^k\, \phi$, on the other hand, ensures that ϕ can be brought into any dashed box with nesting $\leq k$. Compare this with Theorem 5.26.

Example 5.27 We show that the sequent[1] $C(p \vee q)$, $K_1(K_2 p \vee K_2\, \neg p)$, $K_1 \neg K_2\, q \vdash K_1\, p$ is valid in the modal logic KT45^n. That means: if it is common knowledge that $p \vee q$; and agent 1 knows that agent 2 knows whether p is the case and also knows that agent 2 doesn't know that q is true; then agent 1 knows that p is true. See Figure 5.15 for a proof. In line 12, we derived q from $\neg p$ and $p \vee q$. Rather than show the full derivation in propositional logic, which is not the focus here, we summarise by writing 'prop' as the justification for an inference in propositional logic.

[1] In this section we simply write $\vdash$ for $\vdash_{\mathrm{KT45}^n}$, unless indicated otherwise.

5.5.4 Formalising the examples

Now that we have set up the modal logic KT45^n, we can turn our attention to the question of how to represent the wise-men and muddy-children puzzles in this logic. Unfortunately, in spite of its sophistication, our logic is too simple to capture all the nuances of those examples. Although it has connectives for representing different items of knowledge held by different agents, it does not have any temporal aspect, so it cannot directly capture the way in which the agents' knowledge changes as time proceeds. We will overcome this limitation by considering several 'snapshots' during which time is fixed.

The wise-men puzzle Recall that there are three wise men; and it's common knowledge that there are three red hats and two white hats. The king puts a hat on each of the wise men and asks them sequentially whether they know the colour of the hat on their head – they are unable to see their own hat. We suppose the first man says he does not know; then the second says he does not know. We want to prove that, whatever the distribution of hats, the third man now knows his hat is red.

Let p_i mean that man i has a red hat; so $\neg p_i$ means that man i has a white hat. Let Γ be the set of formulas

$$\begin{aligned}
\{&C(p_1 \vee p_2 \vee p_3),\\
&C(p_1 \to K_2\, p_1), C(\neg p_1 \to K_2 \neg p_1),\\
&C(p_1 \to K_3\, p_1), C(\neg p_1 \to K_3 \neg p_1),\\
&C(p_2 \to K_1\, p_2), C(\neg p_2 \to K_1 \neg p_2),\\
&C(p_2 \to K_3\, p_2), C(\neg p_2 \to K_3 \neg p_2),\\
&C(p_3 \to K_1\, p_3), C(\neg p_3 \to K_1 \neg p_3),\\
&C(p_3 \to K_2\, p_3), C(\neg p_3 \to K_2 \neg p_3)\}.
\end{aligned}$$

This corresponds to the initial set-up: it is common knowledge that one of the hats must be red and that each man can see the colour of the other men's hats.

The announcement that the first man doesn't know the colour of his hat amounts to the formula

$$C(\neg K_1\, p_1 \wedge \neg K_1 \neg p_1)$$

and similarly for the second man.

A naive attempt at formalising the wise-men problem might go something like this: we simply prove

$$\Gamma,\ C(\neg K_1\, p_1 \wedge \neg K_1 \neg p_1),\ C(\neg K_2\, p_2 \wedge \neg K_2 \neg p_2)\ \vdash\ K_3\, p_3$$

i.e. if Γ is true and the announcements are made, then the third man knows his hat is red. However, this fails to capture the fact that time passes between the announcements. The fact that $C\neg K_1 p_1$ is true after the first announcement does not mean it is true after some subsequent announcement. For example, if someone announces p_1, then Cp_1 becomes true.

The reason that this formalisation is incorrect is that, although knowledge accrues with time, *lack* of knowledge does not accrue with time. If I know ϕ, then (assuming that ϕ doesn't change) I will know it at the next time-point; but if I do *not* know ϕ, it may be that I *do* know it at the next time point, since I may acquire more knowledge.

To formalise the wise-men problem correctly, we need to break it into two entailments, one corresponding to each announcement. When the first man announces he does not know the colour of his hat, a certain *positive* formula ϕ becomes common knowledge. Our informal reasoning explained that all men could then rule out the state RWW which, given $p_1 \vee p_2 \vee p_3$, led them to the common knowledge of $p_2 \vee p_3$. Thus, ϕ is just $p_2 \vee p_3$ and we need to prove the entailment

Entailment 1. $\Gamma,\ C(\neg K_1 p_1 \wedge \neg K_1 \neg p_1) \vdash C(p_2 \vee p_3)$.

A proof of this sequent can be found in Figure 5.16.

Since $p_2 \vee p_3$ is a positive formula, it persists with time and can be used in conjunction with the second announcement to prove the desired conclusion:

Entailment 2. $\Gamma,\ C(p_2 \vee p_3),\ C(\neg K_2 p_2, \wedge \neg K_2 \neg p_2)\ \vdash\ K_3 p_3$.

This method requires some careful thought: given an announcement of negative information such as a man declaring that he does not know what the colour of his hat is, we need to work out what positive-knowledge formula can be derived from this and such new knowledge has to be sufficient to make even more progress towards solving the puzzle in the next round.

Routine proof segments like those in lines 11–16 of Figure 5.16 may be abbreviated into one step as long as all participating proof rules are recorded. The resulting shorter representation can be seen in Figure 5.17.

In Figure 5.16, notice that the premises in lines 2 and 5 are not used. The premises in lines 2 and 3 stand for any such formula for a given value of i and j, provided $i \neq j$; this explains the inference made in line 8. In Figure 5.18, again notice that the premises in lines 1 and 5 are not used. Observe also that axiom T in conjunction with CK allows us to infer ϕ from any $C\phi$, although we had to split this up into two separate steps in lines 16 and 17. Practical implementations would probably allow for hybrid rules which condense such reasoning into one step.

1	$C(p_1 \vee p_2 \vee p_3)$	premise
2	$C(p_i \to K_j\, p_i)$	premise, $(i \neq j)$
3	$C(\neg p_i \to K_j\, \neg p_i)$	premise, $(i \neq j)$
4	$C\neg K_1\, p_1$	premise
5	$C\neg K_1\, \neg p_1$	premise
6	C	
7	$\neg p_2 \wedge \neg p_3$	assumption
8	$\neg p_2 \to K_1\, \neg p_2$	Ce 3 $(i,j) = (2,1)$
9	$\neg p_3 \to K_1\, \neg p_3$	Ce 3 $(i,j) = (3,1)$
10	$K_1\, \neg p_2 \wedge K_1\, \neg p_3$	prop $7, 8, 9$
11	$K_1\, \neg p_2$	$\wedge$e$_1$ 10
12	$K_1\, \neg p_3$	$\wedge$e$_2$ 10
13	K_1	
14	$\neg p_2$	K_1e 11
15	$\neg p_3$	K_1e 12
16	$\neg p_2 \wedge \neg p_3$	$\wedge$i $14, 15$
17	$p_1 \vee p_2 \vee p_3$	Ce 1
18	p_1	prop 16, 17
19	$K_1\, p_1$	K_1i 13–18
20	$\neg K_1\, p_1$	Ce 4
21	$\bot$	$\neg$e $19, 20$
22	$\neg(\neg p_2 \wedge \neg p_3)$	$\neg$i 7–21
23	$p_2 \vee p_3$	prop 22
24	$C(p_2 \vee p_3)$	Ci 6–23

Figure 5.16. Proof of the sequent 'Entailment 1' for the wise-men puzzle.

The muddy-children puzzle Suppose there are n children. Let p_i mean that the ith child has mud on its forehead. We consider Scenario 2, in which the father announces that one of the children is muddy. Similarly to the case for the wise men, it is common knowledge that each child can see the other children, so it knows whether the others have mud, or not. Thus, for example,

1	$C(p_1 \vee p_2 \vee p_3)$	premise
2	$C(p_i \to K_j\, p_i)$	premise, $(i \neq j)$
3	$C(\neg p_i \to K_j\, \neg p_i)$	premise, $(i \neq j)$
4	$C\neg K_1\, p_1$	premise
5	$C\neg K_1\, \neg p_1$	premise
6	C	
7	$\neg p_2 \wedge \neg p_3$	assumption
8	$\neg p_2 \to K_1\, \neg p_2$	Ce 3 $(i, j) = (2, 1)$
9	$\neg p_3 \to K_1\, \neg p_3$	Ce 3 $(i, j) = (3, 1)$
10	$K_1\, \neg p_2 \wedge K_1\, \neg p_3$	prop $7, 8, 9$
11	K_1	
12	$\neg p_2 \wedge \neg p_3$	$\wedge e_1$, K_1e, $\wedge$i
13	$p_1 \vee p_2 \vee p_3$	Ce 1
14	p_1	prop 12, 13
15	$K_1\, p_1$	K_1i 11–14
16	$\neg K_1\, p_1$	Ce 4
17	$\bot$	$\neg$e $15, 16$
18	$\neg(\neg p_2 \wedge \neg p_3)$	$\neg$i 7–17
19	$p_2 \vee p_3$	prop 18
20	$C(p_2 \vee p_3)$	Ci 6–19

Figure 5.17. A more compact representation of the proof in Figure 5.16.

we have that $C(p_1 \to K_2\, p_1)$, which says that it is common knowledge that, if child 1 is muddy, then child 2 knows this and also $C(\neg p_1 \to K_2\, \neg p_1)$. Let Γ be the collection of formulas:

$$C(p_1 \vee p_2 \vee \cdots \vee p_n)$$

$$\bigwedge_{i \neq j} C(p_i \to K_j\, p_i)$$

$$\bigwedge_{i \neq j} C(\neg p_i \to K_j\, \neg p_i).$$

1	$C(p_1 \vee p_2 \vee p_3)$	premise
2	$C(p_i \to K_j\, p_i)$	premise, $(i \neq j)$
3	$C(\neg p_i \to K_j\, \neg p_i)$	premise, $(i \neq j)$
4	$C\neg K_2\, p_2$	premise
5	$C\neg K_2\, \neg p_2$	premise
6	$C(p_2 \vee p_3)$	premise
7	K_3	
8	$\neg p_3$	assumption
9	$\neg p_3 \to K_2\, \neg p_3$	CK 3 $(i,j) = (3,2)$
10	$K_2\, \neg p_3$	$\to$e $9, 8$
11	K_2	
12	$\neg p_3$	K_2e 10
13	$p_2 \vee p_3$	Ce 6
14	p_2	prop $12, 13$
15	$K_2\, p_2$	K_2i 11−14
16	$K_i\, \neg K_2\, p_2$	CK 4, for each i
17	$\neg K_2\, p_2$	KT 16
18	$\bot$	$\neg$e $15, 17$
19	p_3	PBC 8−18
20	$K_3\, p_3$	K_3i 7−19

Figure 5.18. Proof of the sequent 'Entailment 2' for the wise-men puzzle.

Note that $\bigwedge_{i \neq j} \psi_{(i,j)}$ is a shorthand for the finite conjunction of all formulas $\psi_{(i,j)}$, where i is different from j. Let G be any set of children. We will require formulas of the form

$$\alpha_G \stackrel{\text{def}}{=} \bigwedge_{i \in G} p_i \wedge \bigwedge_{i \notin G} \neg p_i.$$

The formula α_G states that it is precisely the children in G that have muddy foreheads.

1	$\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge p_i \wedge \cdots \wedge \neg p_n$	$\alpha_{\{i\}}$
2	$C(p_1 \vee \cdots \vee p_n)$	in Γ
3	$\neg p_j$	$\wedge$e 1, for each $j \neq i$
4	$\neg p_j \to K_i \neg p_j$	in Γ, for each $j \neq i$
5	$K_i \neg p_j$	$\to$e $4, 3$, for each $j \neq i$
6	$K_i\,(p_1 \vee \cdots \vee p_n)$	CK 2
7	K_i	
8	$p_1 \vee \cdots \vee p_n$	K_i e 6
9	$\neg p_j$	K_i e 5, for each $j \neq i$
10	p_i	prop $9, 8$
11	$K_i\, p_i$	K_i i

Figure 5.19. Proof of the sequent 'Entailment 1' for the muddy-children puzzle.

Suppose now that $k = 1$, i.e. that one child has mud on its forehead. We would like to show that that child knows that it is the one. We prove the following entailment.

Entailment 1. Γ, $\alpha_{\{i\}} \vdash K_i\, p_i$.

This says that, if the actual situation is one in which only one child called i has mud, then that child will know it. Our proof follows exactly the same lines as the intuition: i sees that no other children have mud, but knows that at least one has mud, so knows it must be itself who has a muddy forehead. The proof is given in Figure 5.19.

Note that the comment 'for each $j \neq i$' means that we supply this argument for any such j. Thus, we can form the conjunction of all these inferences which we left implicit in the inference on line 10.

What if there is more than one child with mud? In this case, the children all announce in the first parallel round that they do not know whether they are muddy or not, corresponding to the formula

$$A \overset{\text{def}}{=} C(\neg K_1\, p_1 \wedge \neg K_1 \neg p_1) \wedge \cdots \wedge C(\neg K_n\, p_n \wedge \neg K_n \neg p_n).$$

We saw in the wise-men example that it is dangerous to put the announcement A alongside the premises Γ, because the truth of A, which has negative claims about the children's knowledge, cannot be guaranteed to persist with time. So we seek some positive formula which represents what the children learn upon hearing the announcement. As in the wise-men example, this formula is implicit in the informal reasoning about the muddy children given in Section 5.5.1: if it is common knowledge that there are at least k muddy children, then, after an announcement of the form A, it will be common knowledge that there are at least $k + 1$ muddy children.

Therefore, after the first announcement A, the set of premises is

$$\Gamma, \bigwedge_{1 \le i \le n} C\neg\alpha_{\{i\}}.$$

This is Γ together with the common knowledge that the set of muddy children is not a singleton set.

After the second announcement A, the set of premises becomes

$$\Gamma, \bigwedge_{1 \le i \le n} C\neg\alpha_{\{i\}}, \bigwedge_{i \ne j} C\neg\alpha_{\{i,j\}}$$

which we may write as

$$\Gamma, \bigwedge_{|G| \le 2} C\neg\alpha_G.$$

Please try carefully to understand the notation:

α_G	the set of muddy children is precisely the set G
$\neg\alpha_G$	the set of muddy children is some other set than G
$\bigwedge_{|G| \le k} \neg\alpha_G$	the set of muddy children is of size greater than k.

The entailment corresponding to the second round is:

$$\Gamma, C(\bigwedge_{|G| \le 2} \neg\alpha_G), \alpha_H \vdash \bigwedge_{i \in H} K_i\, p_i, \qquad \text{where } |H| = 3\,.$$

The entailment corresponding to the kth round is:

Entailment 2. Γ, $C(\bigwedge_{|G| \le k} \neg\alpha_G)$, $\alpha_H \vdash \bigwedge_{i \in H} K_i\, p_i$, where $|H| = k + 1$. Please try carefully to understand what this sequent is saying. ‘If all the things in Γ are true and if it is common knowledge that the set of muddy children is not of size less than or equal to k and if actually it is of size $k + 1$, then each of those $k + 1$ children can deduce that they are muddy.’ Notice how this fits with our intuitive account given earlier in this text.

1	α_H	premise
2	$C\neg\alpha_G$	premise as $\lvert G\rvert \leq k$
3	p_j	$\wedge$e 1, for each $j \in G$
4	$\neg p_k$	$\wedge$e 1, for each $k \notin H$
5	$p_j \to K_i\, p_j$	in Γ for each $j \in G$
6	$K_i\, p_j$	$\to$e $5, 4$, for each $j \in G$
7	$\neg p_k \to K_i \neg p_k$	in Γ for each $k \notin H$
8	$K_i \neg p_k$	$\to$e $7, 4$, for each $k \notin H$
9	$K_i \neg\alpha_G$	CK 2
10	K_i	
11	p_j	K_i e 6 ($j \in G$)
12	$\neg p_k$	K_i e 8 ($k \notin H$)
13	$\neg p_i$	assumption
14	α_G	$\wedge$i $11, 12, 13$
15	$\neg\alpha_G$	K_i e 9
16	$\bot$	$\neg$e $14, 15$
17	$\neg\neg p_i$	$\neg$i $13-16$
18	p_i	$\neg\neg$e 17
19	$K_i\, p_i$	K_i i $10-18$

Figure 5.20. The proof of Γ, $C(\neg\alpha_G)$, $\alpha_H \vdash K_i\, p_i$, used to prove 'Entailment 2' for the muddy-children puzzle.

To prove Entailment 2, take any $i \in H$. It is sufficient to prove that

$$\Gamma,\ C(\bigwedge_{|G|\leq k} \neg\alpha_G),\ \alpha_H \vdash K_i\, p_i$$

is valid, as the repeated use of $\wedge$i over all values of i gives us a proof of Entailment 2. Let G be $H - \{i\}$; the proof that $\Gamma,\ C(\neg\alpha_G),\ \alpha_H \vdash K_i\, p_i$ is valid is given in Figure 5.20. Please study this proof in every detail and understand how it is just following the steps taken in the informal proof in Section 5.5.1.

The line 14 of the proof in Figure 5.20 applies several instances of $\wedge$i in sequence and is a legitimate step since the formulas in lines 11–13 had been shown 'for each' element in the respective set.

5.6 Exercises

Exercises 5.1

1. Think about the highly distributed computing environments of today with their dynamic communication and network topology. Come up with several kinds of modes of truth pertaining to statements made about such environments.
2. Let $\mathcal{M}$ be a model of first-order logic and let ϕ range over formulas of first-order logic. Discuss in what sense statements of the form 'Formula ϕ is true in model $\mathcal{M}$.' express a mode of truth.

———

Exercises 5.2

1. Consider the Kripke model $\mathcal{M}$ depicted in Figure 5.5.
   (a) For each of the following, determine whether it holds:
      i. $a \Vdash p$
      ii. $a \Vdash \Box\neg q$
      * iii. $a \Vdash q$
      * iv. $a \Vdash \Box\Box q$
      v. $a \Vdash \Diamond p$
      * vi. $a \Vdash \Box\Diamond\neg q$
      vii. $c \Vdash \Diamond\top$
      viii. $d \Vdash \Diamond\top$
      ix. $d \Vdash \Box\Box q$
      * x. $c \Vdash \Box\bot$
      xi. $b \Vdash \Box\bot$
      xii. $a \Vdash \Diamond\Diamond(p \wedge q) \wedge \Diamond\top$.
   (b) Find for each of the following a world which satisfies it:
      i. $\Box\neg p \wedge \Box\Box\neg p$
      ii. $\Diamond q \wedge \neg\Box q$
      * iii. $\Diamond p \vee \Diamond q$
      * iv. $\Diamond(p \vee \Diamond q)$
      v. $\Box p \vee \Box\neg p$
      vi. $\Box(p \vee \neg p)$.
   (c) For each formula of the previous item, find a world which does not satisfy the formula.
2. Find a Kripke model $\mathcal{M}$ and a formula scheme which is not satisfied in $\mathcal{M}$, but which has true instances in $\mathcal{M}$.

3. Consider the Kripke model $\mathcal{M} = (W, R, L)$ where $W = \{a, b, c, d, e\}$; $R = \{(a, c), (a, e), (b, a), (b, c), (d, e), (e, a)\}$; and $L(a) = \{p\}$, $L(b) = \{p, q\}$, $L(c) = \{p, q\}$, $L(d) = \{q\}$ and $L(e) = \emptyset$.
   (a) Draw a graph for $\mathcal{M}$.
   (b) Investigate which of the formulas in exercise 1(b) on page 350 have a world which satisfies it.
4. (a) Think about what you have to do to decide whether $p \to \Box\Diamond q$ is true in a model.
   * (b) Find a model in which it is true and one in which it is false.
5. For each of the following pairs of formulas, can you find a model and a world in it which distinguishes them, i.e. makes one of them true and one false? In that case, you are showing that they do not entail each other. If you cannot, it might mean that the formulas are equivalent. Justify your answer.
   (a) $\Box p$ and $\Box\Box p$
   (b) $\Box\neg p$ and $\neg\Diamond p$
   (c) $\Box(p \wedge q)$ and $\Box p \wedge \Box q$
   * (d) $\Diamond(p \wedge q)$ and $\Diamond p \wedge \Diamond q$
   (e) $\Box(p \vee q)$ and $\Box p \vee \Box q$
   * (f) $\Diamond(p \vee q)$ and $\Diamond p \vee \Diamond q$
   (g) $\Box(p \to q)$ and $\Box p \to \Box q$
   (h) $\Diamond\top$ and $\top$
   (i) $\Box\top$ and $\top$
   (j) $\Diamond\bot$ and $\bot$.
6. Show that the following formulas of basic modal logic are valid:
   * (a) $\Box(\phi \wedge \psi) \leftrightarrow (\Box\phi \wedge \Box\psi)$
   (b) $\Diamond(\phi \vee \psi) \leftrightarrow (\Diamond\phi \vee \Diamond\psi)$
   * (c) $\Box\top \leftrightarrow \top$
   (d) $\Diamond\bot \leftrightarrow \bot$
   (e) $\Diamond\top \to (\Box\phi \to \Diamond\phi)$
7. Inspect Definition 5.4. We said that we defined $x \Vdash \phi$ by structural induction on ϕ. Is this really correct? Note the implicit definition of a second relation $x \nVdash \phi$. Why is this definition still correct and in what sense does it still rely on structural induction?

Exercises 5.3

1. For which of the readings of $\Box$ in Table 5.7 are the formulas below valid?
   * (a) $(\phi \to \Box\phi) \to (\phi \to \Diamond\phi)$
   (b) $(\Box\phi \to (\phi \wedge \Box\Box\phi \wedge \Diamond\phi)) \to ((\Box\phi \to (\phi \wedge \Box\Box\phi)) \wedge (\Diamond\phi \to \Box\Diamond\phi))$.
2. Dynamic logic: Let P range over the programs of our core language in Chapter 4. Consider a modal logic whose modal operators are $\langle P\rangle$ and $[P]$ for all such programs P. Evaluate such formulas in stores l as in Definition 4.3 (page 264).

The relation $l \vDash \langle P \rangle \, \phi$ holds iff program P has some execution beginning in store l and terminating in a store satisfying ϕ.

* (a) Given that $\neg\langle P\rangle\neg$ equals $[P]$, spell out the meaning of $[P]$.

(b) Say that ϕ is valid iff it holds in all suitable stores l. State the total correctness of a Hoare triple as a validity problem in this modal logic.

3. For all binary relations R below, determine which of the properties reflexive through to total from page 320 apply to R where $R(x, y)$ means that

* (a) x is strictly less than y, where x and y range over all natural numbers $n \geq 1$

(b) x divides y, where x and y range over integers – e.g. 5 divides 15, whereas 7 does not

(c) x is a brother of y

* (d) there exist positive real numbers a and b such that x equals $a \cdot y + b$, where x and y range over real numbers.

* 4. Prove the Fact 5.16.

5. Prove the informal claim made in item 2 of Example 5.12 by structural induction on formulas in (5.1).

6. Prove Theorem 5.17. Use mathematical induction on the length of the sequence of negations and modal operators. Note that this requires a case analysis over the topmost operator other than a negation, or a modality.

7. Prove Theorem 5.14, but for the case in which R is reflexive, or transitive.

8. Find a Kripke model in which all worlds satisfy $\neg p \vee q$, but at least one world does not satisfy $\neg q \vee p$; i.e. show that the scheme $\neg\phi \vee \psi$ is not satisfied.

9. Below you find a list of sequents $\Gamma \vdash \phi$ in propositional logic. Find out whether you can prove them without the use of the rules PBC, LEM and $\neg\neg$e. If you cannot succeed, then try to construct a model $\mathcal{M} = (W, R, L)$ for intuitionistic propositional logic such that one of its worlds satisfies all formulas in Γ, but does not satisfy ϕ. Assuming soundness, this would guarantee that the sequent in question does not have a proof in intuitionistic propositional logic.

* (a) $\vdash (p \to q) \vee (q \to r)$

(b) The proof rule MT: $p \to q, \neg q \vdash \neg p$

(c) $\neg p \vee q \vdash p \to q$

(d) $p \to q \vdash \neg p \vee q$

(e) The proof rule $\neg\neg$e: $\neg\neg p \vdash p$

* (f) The proof rule $\neg\neg$i: $p \vdash \neg\neg p$.

10. Prove that the natural deduction rules for propositional logic without the rules $\neg\neg$e, LEM and PBC are sound for the possible world semantics of intuitionistic propositional logic. Why does this show that the excluded rules cannot be implemented using the remaining ones?

11. Interpreting $\Box\phi$ as 'agent Q believes ϕ,' explain the meaning of the following formula schemes:

(a) $\Box\phi \to \Diamond\phi$

* (b) $\Box\phi \vee \Box\neg\phi$

(c) $\Box(\phi \to \psi) \wedge \Box\phi \to \Box\psi$.

12. In the second row of Table 5.7, we adopted the convention that the future excludes the present. Which formula schemes would be satisfied in that row if instead we adopted the more common convention that the future includes the present?
13. Consider the properties in Table 5.12. Which ones should we accept if we read $\Box$ as
    - * (a) knowledge
    - (b) belief
    - * (c) 'always in the future?'
14. Find a frame which is reflexive, transitive, but not symmetric. Show that your frame does not satisfy the formula $p \to \Box\Diamond p$, by providing a suitable labelling function and choosing a world which refutes $p \to \Box\Diamond p$. Can you find a labelling function and world which does satisfy $p \to \Box\Diamond p$ in your frame?
15. Give two examples of frames which are Euclidean – i.e. their accessibility relation is Euclidean – and two which are not. Explain intuitively why $\Diamond p \to \Box\Diamond p$ holds on the first two, but not on the latter two.
16. For each of the following formulas, find the property of R which corresponds to it.
    - (a) $\phi \to \Box\phi$
    - * (b) $\Box\bot$
    - * (c) $\Diamond\Box\phi \to \Box\Diamond\phi$.

* 17. Find a formula whose corresponding property is density: for all $x, z \in W$ such that $R(x, z)$, there exists $y \in W$ such that $R(x, y)$ and $R(y, z)$.

18. The modal logic KD45 is used to model belief; see Table 5.12 for the axiom schemes D, 4, and 5.
    - (a) Explain how it differs from KT45.
    - (b) Show that $\vDash_{\mathrm{KD45}} \Box p \to \Diamond p$ is valid. What is the significance of this, in terms of knowledge and belief?
    - (c) Explain why the condition of seriality is relevant to belief.
19. Recall Definition 5.7. How would you define $\equiv_L$ for a modal logic L?

Exercises 5.4

1. Find natural deduction proofs for the following sequents over the basic modal logic K.
    - * (a) $\vdash_K \Box(p \to q) \vdash \Box p \to \Box q$
    - (b) $\vdash_K \Box(p \to q) \vdash \Diamond p \to \Diamond q$
    - * (c) $\vdash_K \vdash \Box(p \to q) \land \Box(q \to r) \to \Box(p \to r)$
    - (d) $\vdash_K \Box(p \land q) \vdash \Box p \land \Box q$
    - (e) $\vdash_K \vdash \Diamond\top \to (\Box p \to \Diamond p)$
    - * (f) $\vdash_K \Diamond(p \to q) \vdash \Box p \to \Diamond q$
    - (g) $\vdash_K \Diamond(p \lor q) \vdash \Diamond p \lor \Diamond q$.

2. Find natural deduction proofs for the following, in modal logic KT45.
   (a) $p \to \Box\Diamond p$
   (b) $\Box\Diamond p \leftrightarrow \Diamond p$
   * (c) $\Diamond\Box p \leftrightarrow \Box p$
   (d) $\Box(\Box p \to \Box q) \lor \Box(\Box q \to \Box p)$
   (e) $\Box(\Diamond p \to q) \leftrightarrow \Box(p \to \Box q)$.
3. Study the proofs you gave for the previous exercise to see whether any of these formula schemes could be valid in basic modal logic. Inspect where and how these proofs used the axioms T, 4 and 5 to see whether you can find a counter example, i.e. a Kripke model and a world which does not satisfy the formula.
4. Provide a sketch of an argument which shows that the natural deduction rules for basic modal logic are sound with respect to the semantics $x \Vdash \phi$ over Kripke structures.

Exercises 5.5

1. This exercise is about the wise-men puzzle. Justify your answers.
   (a) Each man is asked the question ‘Do you know the colour of your hat?’ Suppose that the first man says ‘no,’ but the second one says ‘yes.’ Given this information together with the common knowledge, can we infer the colour of his hat?
   (b) Can we predict whether the third man will now answer ‘yes’ or ‘no?’
   (c) What would be the situation if the third man were blind? What about the first man?
2. This exercise is about the muddy-children puzzle. Suppose $k = 4$, say children a, b, c and d have mud on their foreheads. Explain why, before the father’s announcement, it is not common knowledge that someone is dirty.
3. Write formulas for the following:
   (a) Agent 1 knows that p.
   (b) Agent 1 knows that p or q.
   * (c) Agent 1 knows p or agent 1 knows q.
   (d) Agent 1 knows whether p.
   (e) Agent 1 doesn’t know whether p or q.
   (f) Agent 1 knows whether agent 2 knows p.
   * (g) Agent 1 knows whether agent 2 knows whether p.
   (h) No-one knows p.
   (i) Not everyone knows whether p.
   (j) Anyone who knows p knows q.
   * (k) Some people know p but don’t know q.
   (l) Everyone knows someone who knows p.
4. Determine which of the following hold in the Kripke model of Figure 5.13 and justify your answer:

(a) $x_1 \Vdash K_1\, p$

(b) $x_3 \Vdash K_1\, (p \lor q)$

(c) $x_1 \Vdash K_2\, q$

* (d) $x_3 \Vdash E(p \lor q)$

(e) $x_1 \Vdash Cq$

(f) $x_1 \Vdash D_{\{1,3\}}p$

(g) $x_1 \Vdash D_{\{1,2\}}p$

(h) $x_6 \Vdash E\neg q$

* (i) $x_6 \Vdash C\neg q$

(j) $x_6 \Vdash C_{\{3\}}\neg q$.

5. For each of the following formulas, show that it is not valid by finding a Kripke model with a world not satisfying the formula:

   (a) $E_G\, \phi \to E_G E_G\, \phi$

   (b) $\neg E_G\, \phi \to E_G \neg E_G\, \phi$.

   Explain why these two Kripke models show that the union of equivalence relations is not necessarily an equivalence relation.

* 6. Explain why $C_G\, \phi \to C_G C_G\, \phi$ and $\neg C_G\, \phi \to C_G \neg C_G\, \phi$ are valid.

7. Prove the second part of Theorem 5.26.

8. Recall Section 3.7. Can you specify a monotone function over the power set of possible worlds which computes the set of worlds satisfying $C_G\, \phi$? Is this a least, or a greatest, fixed point?

9. Use the natural deduction rules for propositional logic to justify the proof steps below which are only annotated with ‘prop.’

   (a) Line 11 in Figure 5.15.

   (b) Lines 10, 18 and 23 of the proof in Figure 5.16. Of course this requires three separate proofs.

   (c) Line 14 of the proof in Figure 5.18.

   (d) Line 10 of the proof in Figure 5.19.

10. Using the natural deduction rules for KT45^n, prove the validity of

    (a) $K_i\, (p \land q) \leftrightarrow K_i\, p \land K_i\, q$

    (b) $C(p \land q) \leftrightarrow Cp \land Cq$

    * (c) $K_i\, Cp \leftrightarrow Cp$

    (d) $C\, K_i\, p \leftrightarrow Cp$

    * (e) $\neg\phi \to K_i \neg K_i\, \phi$.

    Explain what this formula means in terms of knowledge. Do you believe it?

    (f) $\neg\phi \to K_1 K_2 \neg K_2 K_1\, \phi$

    * (g) $\neg K_1 \neg K_1 \phi \leftrightarrow K_1\, \phi$.

11. Do a natural deduction proof for a simpler version of the wise-men problem: There are two wise men; as usual, they can see each other’s hats but not their own. It is common knowledge that there’s only one white hat available and two red ones. So at least one of the men is wearing a red one. Man 1 informs the second that he doesn’t know which hat he is wearing. Man 2 says, ‘Aha, then I must be wearing a red hat.’

(a) Justify man 2's conclusion informally.

(b) Let p_1, p_2 respectively, mean man 1, 2 respectively, is wearing a red hat. So $\neg p_1, \neg p_2$ mean they (respectively) are wearing a white one. Informally justify each of the following premises in terms of the description of the problem:

i. $K_2 K_1 (p_1 \vee p_2)$

ii. $K_2(\neg p_2 \to K_1 \neg p_2)$

iii. $K_2 \neg K_1 p_1$.

(c) Using natural deduction, prove from these premises that $K_2 p_2$.

(d) Show that the third premise was essential, by exhibiting a model/world which satisfies the first two, but not the conclusion.

(e) Now is it easy to answer questions like 'If man 2 were blind would he still be able to tell?' and 'if man 1 were blind, would man 2 still be able to tell?'?

12. Recall our informal discussion on positive-knowledge formulas and negative-knowledge formulas. Give formal definitions of these notions.

5.7 Bibliographic notes

The first systematic approaches to modal logic were made by C. I. Lewis in the 1950s. The possible-worlds approach, which greatly simplified modal logic and is now almost synonymous with it, was invented by S. Kripke. Books devoted to modal logic include [Che80, Gol87, Pop94], where extensive references to the literature may be found. All these books discuss the soundness and completeness of proof calculi for modal logics. They also investigate which modal logics have the *finite-model property*: if a sequent does not have a proof, there is a finite model which demonstrates that. Not all modal logics enjoy this property, which is important for decidability. Intuitionistic propositional logic has the finite-model property; an animation which generates such finite models (called PORGI) is available from A. Stoughton's website[2].

The idea of using modal logic to reason about knowledge is due to J. Hintikka. A great deal of work on applying modal logic to multi-agent systems has been done in [FHMV95] and [MvdH95] and other work by those authors. Many examples in this chapter are taken from this literature (some of them are attributed to other people there), though our treatment of them is original.

The natural deduction proof system for modal logic presented in this chapter is based on ideas in [Fit93].

[2] www.cis.ksu.edu/~allen/porgi.html

An application of the modal logic KT4 (more precisely, its fragment without negation) as a type system for staged computation in a functional programming language can be found in [DP96].

We should stress that our framework was deliberately ‘classical;’ the thesis [Sim94] is a good source for discussions of intuitionistic modal logics; it also contains a gentle introduction to basic first-order modal logic.

6
Binary decision diagrams

6.1 Representing boolean functions

Boolean functions are an important descriptive formalism for many hardware and software systems, such as synchronous and asynchronous circuits, reactive systems and finite-state programs. Representing those systems in a computer in order to reason about them requires an efficient representation for boolean functions. We look at such a representation in this chapter and describe in detail how the systems discussed in Chapter 3 can be verified using the representation.

Definition 6.1 A boolean variable x is a variable ranging over the values 0 and 1. We write $x_1, x_2, \dots$ and $x, y, z, \dots$ to denote boolean variables. We define the following functions on the set $\{0, 1\}$:

- $\overline{0} \overset{\text{def}}{=} 1$ and $\overline{1} \overset{\text{def}}{=} 0$;
- $x \cdot y \overset{\text{def}}{=} 1$ if x and y have value 1; otherwise $x \cdot y \overset{\text{def}}{=} 0$;
- $x + y \overset{\text{def}}{=} 0$ if x and y have value 0; otherwise $x + y \overset{\text{def}}{=} 1$;
- $x \oplus y \overset{\text{def}}{=} 1$ if exactly one of x and y equals 1.

A boolean function f of n arguments is a function from $\{0, 1\}^n$ to $\{0, 1\}$. We write $f(x_1, x_2, \dots, x_n)$, or $f(V)$, to indicate that a syntactic representation of f depends on the boolean variables in V only.

Note that $\cdot$, $+$ and $\oplus$ are boolean functions with two arguments, whereas $\bar{}$ is a boolean function that takes one argument. The binary functions $\cdot$, $+$ and $\oplus$ are written in infix notation instead of prefix; i.e. we write $x + y$ instead of $+(x, y)$, etc.

Example 6.2 In terms of the four functions above, we can define other boolean functions such as

(1) $f(x, y) \stackrel{\text{def}}{=} x \cdot (y + \overline{x})$
(2) $g(x, y) \stackrel{\text{def}}{=} x \cdot y + (1 \oplus \overline{x})$
(3) $h(x, y, z) \stackrel{\text{def}}{=} x + y \cdot (x \oplus \overline{y})$
(4) $k() \stackrel{\text{def}}{=} 1 \oplus (0 \cdot \overline{1})$.

6.1.1 Propositional formulas and truth tables

Truth tables and *propositional formulas* are two different representations of boolean functions. In propositional formulas, $\wedge$ denotes $\cdot$, $\vee$ denotes $+$, $\neg$ denotes $\bar{}$ and $\top$ and $\bot$ denote 1 and 0, respectively.

Boolean functions are represented by truth tables in the obvious way; for example, the function $f(x, y) \stackrel{\text{def}}{=} \overline{x + y}$ is represented by the truth table on the left:

x	y	$f(x, y)$
1	1	0
0	1	0
1	0	0
0	0	1

p	q	ϕ
T	T	F
F	T	F
T	F	F
F	F	T

On the right, we show the same truth table using the notation of Chapter 1; a formula having this truth table is $\neg(p \vee q)$. In this chapter, we may mix these two notational systems of boolean formulas and formulas of propositional logic whenever it is convenient. You should be able to translate expressions easily from one notation to the other and vice versa.

As representations of boolean functions, propositional formulas and truth tables have different advantages and disadvantages. Truth tables are very space-inefficient: if one wanted to model the functionality of a sequential circuit by a boolean function of 100 variables (a small chip component would easily require this many variables), then the truth table would require 2^{100} (which is more than 10^{30}) lines. Alas, there is not enough storage space (whether paper or particle) in the universe to record the information of 2^{100} different bit vectors of length 100. Although they are space inefficient, operations on truth tables are simple. Once you have computed a truth table, it is easy to see whether the boolean function represented is satisfiable: you just look to see if there is a 1 in the last column of the table.

Comparing whether two truth tables represent the same boolean function also seems easy: assuming the two tables are presented with the same order of valuations, we simply check that they are identical. Although these operations seem simple, however, they are computationally intractable because of the fact that the number of lines in the truth table is exponential in the number of variables. Checking satisfiability of a function with n atoms requires of the order of 2^n operations if the function is represented as a truth table. We conclude that checking satisfiability and equivalence is highly inefficient with the truth-table representation.

Representation of boolean functions by propositional formulas is slightly better. Propositional formulas often provide a wonderfully compact and efficient presentation of boolean functions. A formula with 100 variables might only be about 200–300 characters long. However, deciding whether an arbitrary propositional formula is satisfiable is a famous problem in computer science: no efficient algorithms for this task are known, and it is strongly suspected that there aren't any. Similarly, deciding whether two arbitrary propositional formulas f and g denote the same boolean function is suspected to be exponentially expensive.

It is straightforward to see how to perform the boolean operations $\cdot$, $+$, $\oplus$ and $\bar{\ }$ on these two representations. In the case of truth tables, they involve applying the operation to each line; for example, given truth tables for f and g over the same set of variables (and in the same order), the truth table for $f \oplus g$ is obtained by applying $\oplus$ to the truth value of f and g in each line. If f and g do not have the same set of arguments, it is easy to pad them out by adding further arguments. In the case of representation by propositional formulas, the operations $\cdot$, $\oplus$, etc., are simply syntactic manipulations. For example, given formulas ϕ and ψ representing the functions f and g, the formulas representing $f \cdot g$ and $f \oplus g$ are, respectively, $\phi \land \psi$ and $(\phi \land \neg\psi) \lor (\neg\phi \land \psi)$.

We could also consider representing boolean functions by various subclasses of propositional formulas, such as conjunctive and disjunctive normal forms. In the case of disjunctive normal form (DNF, in which a formula is a disjunction of conjunctions of literals), the representation is sometimes compact, but in the worst cases it can be very lengthy. Checking satisfiability is a straightforward operation, however, because it is sufficient to find a disjunct which does not have two complementary literals. Unfortunately, there is not a similar way of checking validity. Performing $+$ on two formulas in DNF simply involves inserting $\lor$ between them. Performing $\cdot$ is more complicated; we cannot simply insert $\land$ between the two formulas, because the result will not in general be in DNF, so we have to perform lengthy applications of the distributivity rule $\phi \land (\psi_1 \lor \psi_2) \equiv (\phi \land \psi_1) \lor (\phi \land \psi_1)$. Computing the negation of a DNF formula is also expensive. The DNF formula ϕ may be

Representation of		test for		boolean operations		
boolean functions	compact?	satisf'ty	validity	$\cdot$	$+$	$^-$
Prop. formulas	often	hard	hard	easy	easy	easy
Formulas in DNF	sometimes	easy	hard	hard	easy	hard
Formulas in CNF	sometimes	hard	easy	easy	hard	hard
Ordered truth tables	never	hard	hard	hard	hard	hard
Reduced OBDDs	often	easy	easy	medium	medium	easy

Figure 6.1. Comparing efficiency of five representations of boolean formulas.

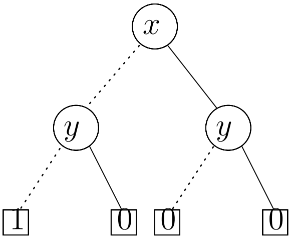


Figure 6.2. An example of a binary decision tree.

quite short, whereas the length of the disjunctive normal form of $\neg\phi$ can be exponential in the length of ϕ.

The situation for representation in conjunctive normal form is the dual. A summary of these remarks is contained in Figure 6.1 (for now, please ignore the last row).

6.1.2 Binary decision diagrams

Binary decision diagrams (BDDs) are another way of representing boolean functions. A certain class of such diagrams will provide the implementational framework for our symbolic model-checking algorithm. Binary decision diagrams were first considered in a simpler form called *binary decision trees.* These are trees whose non-terminal nodes are labelled with boolean variables $x, y, z, \ldots$ and whose terminal nodes are labelled with either 0 or 1. Each non-terminal node has two edges, one dashed line and one solid line. In Figure 6.2 you can see such a binary decision tree with two layers of variables x and y.

Definition 6.3 Let T be a finite binary decision tree. Then T determines a unique boolean function of the variables in non-terminal nodes, in the following way. Given an assignment of 0s and 1s to the boolean variables

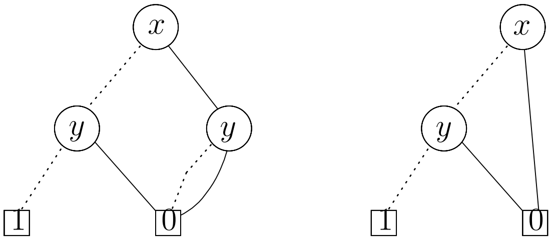


Figure 6.3. (a) Sharing the terminal nodes of the binary decision tree in Figure 6.2; (b) further optimisation by removing a redundant decision point.

occurring in T, we start at the root of T and take the dashed line whenever the value of the variable at the current node is 0; otherwise, we travel along the solid line. The function value is the value of the terminal node we reach.

For example, the binary decision tree of Figure 6.2 represents a boolean function $f(x, y)$. To find $f(0, 1)$, start at the root of the tree. Since the value of x is 0 we follow the dashed line out of the node labelled x and arrive at the leftmost node labelled y. Since y's value is 1, we follow the solid line out of that y-node and arrive at the leftmost terminal node labelled 0. Thus, $f(0, 1)$ equals 0. In computing $f(0, 0)$, we similarly travel down the tree, but now following two dashed lines to obtain 1 as a result. You can see that the two other possibilities result in reaching the remaining two terminal nodes labelled 0. Thus, this binary decision tree computes the function $f(x, y) \overset{\text{def}}{=} \overline{x + y}$.

Binary decision trees are quite close to the representation of boolean functions as truth tables as far as their sizes are concerned. If the root of a binary decision tree is an x-node then it has two subtrees (one for the value of x being 0 and another one for x having value 1). So if f depends on n boolean variables, the corresponding binary decision tree will have at least $2^{n+1} - 1$ nodes (see exercise 5 on page 399). Since f's truth table has 2^n lines, we see that decision trees as such are not a more compact representation of boolean functions. However, binary decision trees often contain some redundancy which we can exploit.

Since 0 and 1 are the only terminal nodes of binary decision trees, we can optimise the representation by having pointers to just one copy of 0 and one copy of 1. For example, the binary decision tree in Figure 6.2 can be optimised in this way and the resulting structure is depicted in Figure 6.3(a). Note that we saved storage space for two redundant terminal 0-nodes, but that we still have as many edges (pointers) as before.

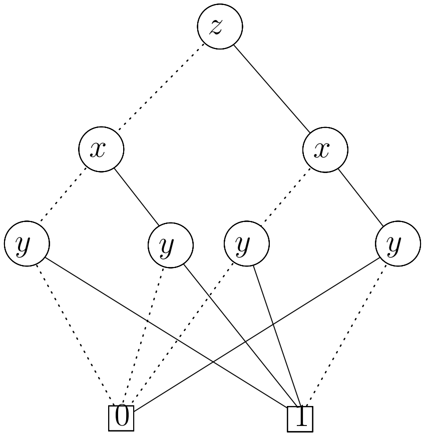


Figure 6.4. A BDD with duplicated subBDDs.

A second optimisation we can do is to remove unnecessary decision points in the tree. In Figure 6.3(a), the right-hand y is unnecessary, because we go to the same place whether it is 0 or 1. Therefore the structure could be further reduced, to the one shown on the right, (b).

All these structures are examples of *binary decision diagrams* (BDDs). They are more general than binary decision trees; the sharing of the leaves means they are not trees. As a third optimisation, we also allow subBDDs to be shared. A subBDD is the part of a BDD occurring below a given node. For example, in the BDD of Figure 6.4, the two inner y-nodes perform the same role, because the subBDDs below them have the same structure. Therefore, one of them could be removed, resulting in the BDD in Figure 6.5(a). Indeed, the left-most y-node could also be merged with the middle one; then the x-node above both of them would become redundant. Removing it would result in the BDD on the right of Figure 6.5.

To summarise, we encountered three different ways of reducing a BDD to a more compact form:

C1. Removal of duplicate terminals. If a BDD contains more than one terminal 0-node, then we redirect all edges which point to such a 0-node to just one of them. We proceed in the same way with terminal nodes labelled with 1.

C2. Removal of redundant tests. If both outgoing edges of a node n point to the same node m, then we eliminate that node n, sending all its incoming edges to m.

C3. Removal of duplicate non-terminals. If two distinct nodes n and m in the BDD are the roots of structurally identical subBDDs, then we

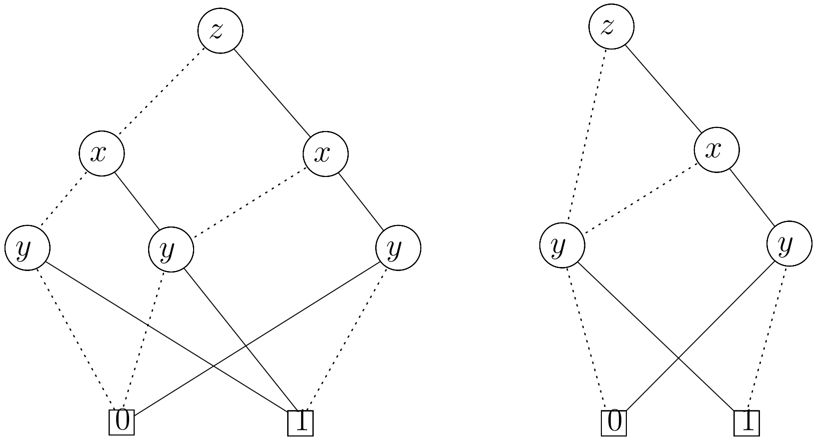


Figure 6.5. The BDD of Figure 6.4: (a) after removal of one of the duplicate y-nodes; (b) after removal of another duplicate y-node and then a redundant x-decision point.

eliminate one of them, say m, and redirect all its incoming edges to the other one.

Note that C1 is a special case of C3. In order to define BDDs precisely, we need a few auxiliary notions.

Definition 6.4 A directed graph is a set G and a binary relation $\rightarrow$ on G: $\rightarrow \subseteq G \times G$. A cycle in a directed graph is a finite path in that graph that begins and ends at the same node, i.e. a path of the form $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$. A directed acyclic graph (dag) is a directed graph that does not have any cycles. A node of a dag is initial if there are no edges pointing to that node. A node is called terminal if there are no edges out of that node.

The directed graph in Figure 3.3 on page 179 has cycles, for example the cycle $s_0 \rightarrow s_1 \rightarrow s_0$, and is not a dag. If we interpret the links in BDDs (whether solid or dashed) as always going in a downwards direction, then the BDDs of this chapter are also directed graphs. They are also acyclic and have a unique initial node. The optimisations C1–C3 preserve the property of being a dag; and fully reduced BDDs have precisely two terminal nodes. We now formally define BDDs as certain kinds of dags:

Definition 6.5 A binary decision diagram (BDD) is a finite dag with a unique initial node, where all terminal nodes are labelled with 0 or 1 and all non-terminal nodes are labelled with a boolean variable. Each

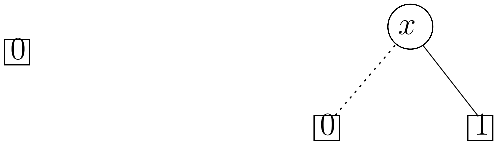


Figure 6.6. The BDDs (a) B_0, representing the constant 0 boolean function; similarly, the BDD B_1 has only one node 1 and represents the constant 1 boolean function; and (b) B_x, representing the boolean variable x.

non-terminal node has exactly two edges from that node to others: one labelled 0 and one labelled 1 (we represent them as a dashed line and a solid line, respectively).

A BDD is said to be reduced if none of the optimisations C1–C3 can be applied (i.e. no more reductions are possible).

All the decision structures we have seen in this chapter (Figures 6.2–6.5) are BDDs, as are the constant functions B_0 and B_1, and the function B_x from Figure 6.6. If B is a BDD where $V = \{x_1, x_2, \dots, x_n\}$ is the set of labels of non-terminal nodes, then B determines a boolean function $f(V)$ in the same way as binary decision trees (see Definition 6.3): given an assignment of 0s and 1s to the variables in V, we compute the value of f by starting with the unique initial node. If its variable has value 0, we follow the dashed line; otherwise we take the solid line. We continue for each node until we reach a terminal node. Since the BDD is finite by definition, we eventually reach a terminal node which is labelled with 0 or 1. That label is the result of f for that particular assignment of truth values.

The definition of a BDD does not prohibit that a boolean variable occur more than once on a path in the dag. For example, consider the BDD in Figure 6.7.

Such a representation is wasteful, however. The solid link from the leftmost x to the 1-terminal is never taken, for example, because one can only get to that x-node when x has value 0.

Thanks to the reductions C1–C3, BDDs can often be quite compact representations of boolean functions. Let us consider how to check satisfiability and perform the boolean operations on functions represented as BDDs. A BDD represents a satisfiable function if a 1-terminal node is reachable from the root along a *consistent path* in a BDD which represents it. A consistent path is one which, for every variable, has only dashed lines or only solid lines leaving nodes labelled by that variable. (In other words, we cannot assign

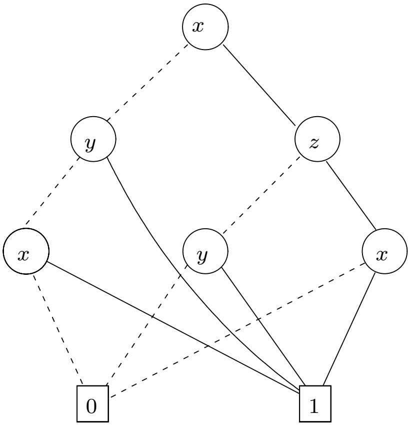


Figure 6.7. A BDD where some boolean variables occur more than once on an evaluation path.

a variable the values 0 and 1 simultaneously.) Checking validity is similar, but we check that no 0-terminal is reachable by a consistent path.

The operations $\cdot$ and $+$ can be performed by 'surgery' on the component BDDs. Given BDDs B_f and B_g representing boolean functions f and g, a BDD representing $f \cdot g$ can be obtained by taking the BDD f and replacing all its 1-terminals by B_g. To see why this is so, consider how to get to a 1-terminal in the resulting BDD. You have to satisfy the requirements for getting to a 1 imposed by both of the BDDs. Similarly, a BDD for $f + g$ can be obtained by replacing all 0 terminals of B_f by B_g. Note that these operations are likely to generate BDDs with multiple occurrences of variables along a path. Later, in Section 6.2, we will see definitions of $+$ and $\cdot$ on BDDs that don't have this undesirable effect.

The complementation operation $\bar{\ }$ is also possible: a BDD representing $\overline{f}$ can be obtained by replacing all 0-terminals in B_f by 1-terminals and vice versa. Figure 6.8 shows the complement of the BDD in Figure 6.2.

6.1.3 Ordered BDDs

We have seen that the representation of boolean functions by BDDs is often compact, thanks to the sharing of information afforded by the reductions C1–C3. However, BDDs with multiple occurrences of a boolean variable along a path seem rather inefficient. Moreover, there seems no easy way to test for equivalence of BDDs. For example, the BDDs of Figures 6.7 and 6.9 represent the same boolean function (the reader should check this). Neither of them can be optimised further by applying the rules C1–C3. However,

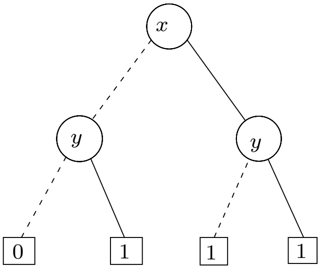


Figure 6.8. The complement of the BDD in Figure 6.2.

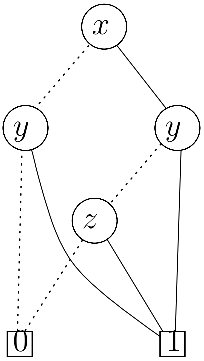


Figure 6.9. A BDD representing the same function as the BDD of Figure 6.7, but having the variable ordering $[x, y, z]$.

testing whether they denote the same boolean function seems to involve as much computational effort as computing the entire truth table for $f(x, y, z)$.

We can improve matters by imposing an ordering on the variables occurring along any path. We then adhere to that same ordering for all the BDDs we manipulate.

Definition 6.6 Let $[x_1, \dots, x_n]$ be an ordered list of variables without duplications and let B be a BDD all of whose variables occur somewhere in the list. We say that B has the ordering $[x_1, \dots, x_n]$ if all variable labels of B occur in that list and, for every occurrence of x_i followed by x_j along any path in B, we have $i < j$.

An ordered BDD (OBDD) is a BDD which has an ordering for some list of variables.

Note that the BDDs of Figures 6.3(a,b) and 6.9 are ordered (with ordering $[x, y]$). We don't insist that every variable in the list is used in the paths. Thus, the OBDDs of Figures 6.3 and 6.9 have the ordering $[x, y, z]$ and so

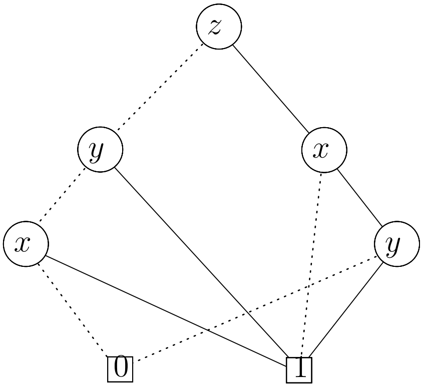


Figure 6.10. A BDD which does not have an ordering of variables.

does any list having x, y and z in it in that order, such as $[u, x, y, v, z, w]$ and $[x, u, y, z]$. Even the BDDs B_0 and B_1 in Figure 6.6 are OBDDs, a suitable ordering list being the empty list (there are no variables), or indeed *any* list. The BDD B_x of Figure 6.6(b) is also an OBDD, with any list containing x as its ordering.

The BDD of Figure 6.7 is not ordered. To see why this is so, consider the path taken if the values of x and y are 0. We begin with the root, an x-node, and reach a y-node and then an x-node again. Thus, no matter what list arrangement we choose (remembering that no double occurrences are allowed), this path violates the ordering condition. Another example of a BDD that is not ordered can be seen in Figure 6.10. In that case, we cannot find an order since the path for $(x, y, z) \Rightarrow (0, 0, 0)$ – meaning that x, y and z are assigned 0 – shows that y needs to occur before x in such a list, whereas the path for $(x, y, z) \Rightarrow (1, 1, 1)$ demands that x be before y.

It follows from the definition of OBDDs that one cannot have multiple occurrences of any variable along a path.

When operations are performed on two OBDDs, we usually require that they have *compatible variable orderings.* The orderings of B_1 and B_2 are said to be compatible if there are no variables x and y such that x comes before y in the ordering of B_1 and y comes before x in the ordering of B_2. This commitment to an ordering gives us a unique representation of boolean functions as OBDDs. For example, the BDDs in Figures 6.8 and 6.9 have compatible variable orderings.

Theorem 6.7 The reduced OBDD representing a given function f is unique. That is to say, let B and B' be two reduced OBDDs with compatible variable orderings. If B and B represent the same boolean function, then they have identical structure.

In other words, with OBDDs we cannot get a situation like the one encountered earlier, in which we have two distinct reduced BDDs which represent the same function, provided that the orderings are compatible. It follows that checking equivalence of OBDDs is immediate. Checking whether two OBDDs (having compatible orderings) represent the same function is simply a matter of checking whether they have the same structure[1].

A useful consequence of the theorem above is that, if we apply the reductions C1–C3 to an OBDD until no further reductions are possible, then we are guaranteed that the result is always the same reduced OBDD. The order in which we applied the reductions does not matter. We therefore say that OBDDs have a *canonical form*, namely their unique reduced OBDD. Most other representations (conjunctive normal forms, etc.) do not have canonical forms.

The algorithms for $\cdot$ and $+$ for BDDs, presented in Section 6.1.2, won't work for OBDDs as they may introduce multiple occurrences of the same variable on a path. We will soon develop more sophisticated algorithms for these operations on OBDDs, which exploit the compatible ordering of variables in paths.

OBDDs allow compact representations of certain classes of boolean functions which only have exponential representations in other systems, such as truth tables and conjunctive normal forms. As an example consider the *even parity function* $f_{\mathrm{even}}(x_1, x_2, \dots, x_n)$ which is defined to be 1 if there is an even number of variables x_i with value 1; otherwise, it is defined to be 0. Its representation as an OBDD requires only $2n + 1$ nodes. Its OBDD for $n = 4$ and the ordering $[x_1, x_2, x_3, x_4]$ can be found in Figure 6.11.

The impact of the chosen variable ordering The size of the OBDD representing the parity functions is independent of the chosen variable ordering. This is because the parity functions are themselves independent of the order of variables: swapping the values of any two variables does not change the value of the function; such functions are called symmetric.

However, in general the chosen variable ordering makes a significant difference to the size of the OBDD representing a given function. Consider the boolean function $(x_1 + x_2) \cdot (x_3 + x_4) \cdot \dots \cdot (x_{2n-1} + x_{2n})$; it corresponds to a propositional formula in conjunctive normal form. If we choose the

[1] In an implementation this will amount to checking whether two pointers are equal.

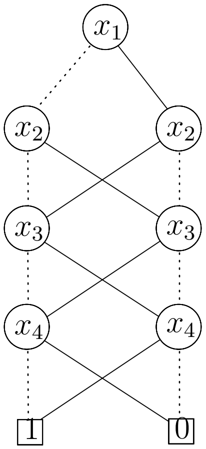


Figure 6.11. An OBDD for the even parity function for four bits.

'natural' ordering $[x_1, x_2, x_3, x_4, \dots]$, then we can represent this function as an OBDD with $2n + 2$ nodes. Figure 6.12 shows the resulting OBDD for $n = 3$. Unfortunately, if we choose instead the ordering

$$[x_1, x_3, \dots, x_{2n-1}, x_2, x_4, \dots, x_{2n}]$$

the resulting OBDD requires 2^{n+1} nodes; the OBDD for $n = 3$ can be seen in Figure 6.13.

The sensitivity of the size of an OBDD to the particular variable ordering is a price we pay for all the advantages that OBDDs have over BDDs. Although finding the optimal ordering is itself a computationally expensive problem, there are good heuristics which will usually produce a fairly good ordering. Later on we return to this issue in discussions of applications.

The importance of canonical representation The importance of having a canonical form for OBDDs in conjunction with an efficient test for deciding whether two reduced OBDDs are isomorphic cannot be overestimated. It allows us to perform the following tests:

Absence of redundant variables. If the value of the boolean function $f(x_1, x_2, \dots, x_n)$ does not depend on the value of x_i, then any reduced OBDD which represents f does not contain any x_i-node.

Test for semantic equivalence. If two functions $f(x_1, x_2, \dots, x_n)$ and $g(x_1, x_2, \dots, x_n)$ are represented by OBDDs B_f, respectively B_g, with a compatible ordering of variables, then we can efficiently decide whether f and g are semantically equivalent. We reduce B_f and B_g (if necessary); f

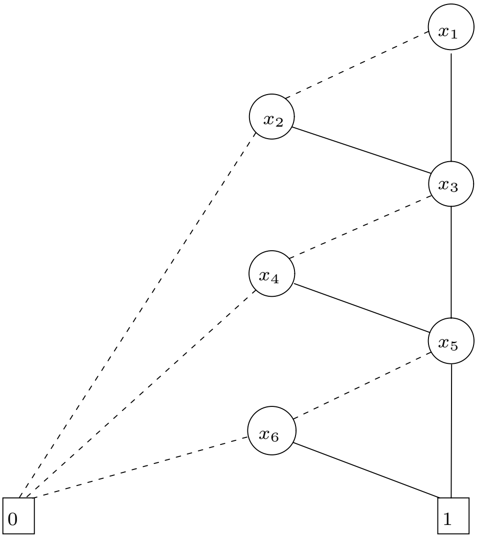


Figure 6.12. The OBDD for $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_2, x_3, x_4, x_5, x_6]$.

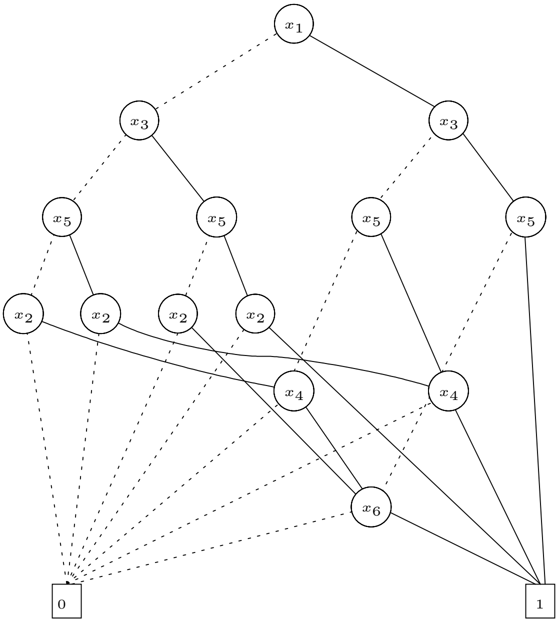


Figure 6.13. Changing the ordering may have dramatic effects on the size of an OBDD: the OBDD for $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_3, x_5, x_2, x_4, x_6]$.

and g denote the same boolean functions if, and only if, the reduced OBDDs have identical structure.

Test for validity. We can test a function $f(x_1, x_2, \ldots, x_n)$ for validity (i.e. f always computes 1) in the following way. Compute a reduced OBDD for f. Then f is valid if, and only if, its reduced OBDD is B_1.

Test for implication. We can test whether $f(x_1, x_2, \ldots, x_n)$ implies $g(x_1, x_2, \ldots, x_n)$ (i.e. whenever f computes 1, then so does g) by computing the reduced OBDD for $f \cdot \overline{g}$. This is B_0 iff the implication holds.

Test for satisfiability. We can test a function $f(x_1, x_2, \ldots, x_n)$ for satisfiability (f computes 1 for at least one assignment of 0 and 1 values to its variables). The function f is satisfiable iff its reduced OBDD is not B_0.

6.2 Algorithms for reduced OBDDs

6.2.1 The algorithm `reduce`

The reductions C1–C3 are at the core of any serious use of OBDDs, for whenever we construct a BDD we will want to convert it to its reduced form. In this section, we describe an algorithm `reduce` which does this efficiently for ordered BDDs.

If the ordering of B is $[x_1, x_2, \ldots, x_l]$, then B has at most $l + 1$ layers. The algorithm `reduce` now traverses B layer by layer in a bottom-up fashion, beginning with the terminal nodes. In traversing B, it assigns an integer label $\mathrm{id}(n)$ to each node n of B, in such a way that the subOBDDs with root nodes n and m denote the same boolean function if, and only if, $\mathrm{id}(n)$ equals $\mathrm{id}(m)$.

Since `reduce` starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label (compare with reduction C1). Similarly, the 1-nodes all get the next label, say #1.

Now let us inductively assume that `reduce` has already assigned integer labels to all nodes of a layer $> i$ (i.e. all terminal nodes and x_j-nodes with $j > i$). We describe how nodes of layer i (i.e. x_i-nodes) are being handled.

Definition 6.8 Given a non-terminal node n in a BDD, we define $\mathrm{lo}(n)$ to be the node pointed to via the dashed line from n. Dually, $\mathrm{hi}(n)$ is the node pointed to via the solid line from n.

Let us describe how the labelling is done. Given an x_i-node n, there are three ways in which it may get its label:

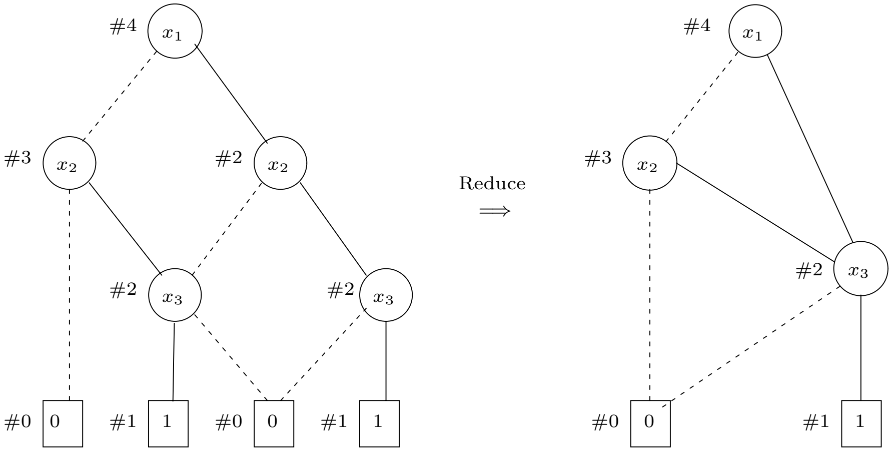


Figure 6.14. An example execution of the algorithm `reduce`.

- If the label id(lo(n)) is the same as id(hi(n)), then we set id(n) to be that label. That is because the boolean function represented at n is the same function as the one represented at lo(n) and hi(n). In other words, node n performs a redundant test and can be eliminated by reduction C2.
- If there is another node m such that n and m have the same variable x_i, and id(lo(n)) = id(lo(m)) and id(hi(n)) = id(hi(m)), then we set id(n) to be id(m). This is because the nodes n and m compute the same boolean function (compare with reduction C3).
- Otherwise, we set id(n) to the next unused integer label.

Note that only the last case creates a new label. Consider the OBDD in left side of Figure 6.14; each node has an integer label obtained in the manner just described. The algorithm `reduce` then finishes by redirecting edges bottom-up as outlined in C1–C3. The resulting reduced OBDD is in right of Figure 6.14. Since there are efficient bottom-up traversal algorithms for dags, `reduce` is an efficient operation in the number of nodes of an OBDD.

6.2.2 The algorithm `apply`

Another procedure at the heart of OBDDs is the algorithm `apply`. It is used to implement operations on boolean functions such as $+$, $\cdot$, $\oplus$ and complementation (via $f \oplus 1$). Given OBDDs B_f and B_g for boolean formulas f and g, the call `apply` (op, B_f, B_g) computes the reduced OBDD of the boolean formula $f \,\mathrm{op}\, g$, where op denotes any function from $\{0,1\} \times \{0,1\}$ to $\{0,1\}$.

The intuition behind the `apply` algorithm is fairly simple. The algorithm operates recursively on the structure of the two OBDDs:

1. let v be the variable highest in the ordering (=leftmost in the list) which occurs in B_f or B_g.
2. split the problem into two subproblems for v being 0 and v being 1 and solve recursively;
3. at the leaves, apply the boolean operation op directly.

The result will usually have to be reduced to make it into an OBDD. Some reduction can be done 'on the fly' in step 2, by avoiding the creation of a new node if both branches are equal (in which case return the common result), or if an equivalent node already exists (in which case, use it).

Let us make all this more precise and detailed.

Definition 6.9 Let f be a boolean formula and x a variable.

1. We denote by $f[0/x]$ the boolean formula obtained by replacing all occurrences of x in f by 0. The formula $f[1/x]$ is defined similarly. The expressions $f[0/x]$ and $f[1/x]$ are called restrictions of f.
2. We say that two boolean formulas f and g are semantically equivalent if they represent the same boolean function (with respect to the boolean variables that they depend upon). In that case, we write $f \equiv g$.

For example, if $f(x, y) \stackrel{\text{def}}{=} x \cdot (y + \overline{x})$, then $f[0/x](x, y)$ equals $0 \cdot (y + \overline{0})$, which is semantically equivalent to 0. Similarly, $f[1/y](x, y)$ is $x \cdot (1 + \overline{x})$, which is semantically equivalent to x.

Restrictions allow us to perform recursion on boolean formulas, by decomposing boolean formulas into simpler ones. For example, if x is a variable in f, then f is equivalent to $\overline{x} \cdot f[0/x] + x \cdot f[1/x]$. To see this, consider the case $x = 0$; the expression computes to $f[0/x]$. When $x = 1$ it yields $f[1/x]$. This observation is known as the *Shannon expansion*, although it can already be found in G. Boole's book *'The Laws of Thought'* from 1854.

Lemma 6.10 (Shannon expansion) For all boolean formulas f and all boolean variables x (even those not occurring in f) we have

$$f \equiv \overline{x} \cdot f[0/x] + x \cdot f[1/x]. \tag{6.1}$$

The function `apply` is based on the Shannon expansion for $f \operatorname{op} g$:

$$f \operatorname{op} g = \overline{x_i} \cdot (f[0/x_i] \operatorname{op} g[0/x_i]) + x_i \cdot (f[1/x_i] \operatorname{op} g[1/x_i]). \tag{6.2}$$

This is used as a control structure of `apply` which proceeds from the roots

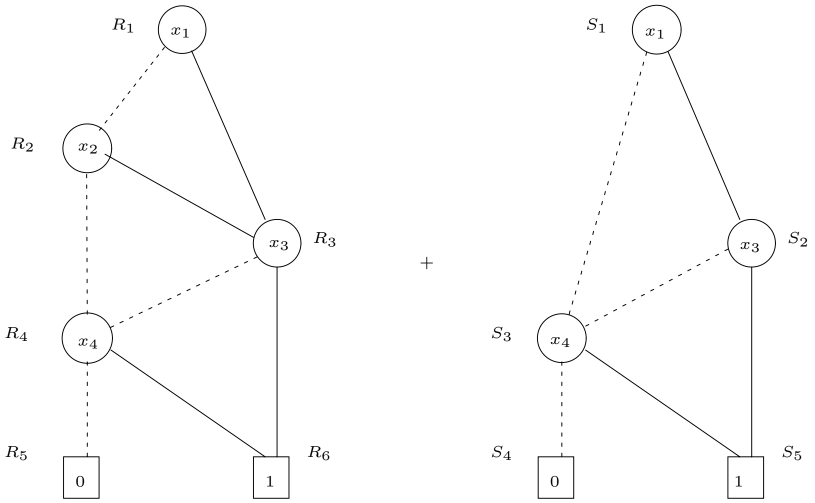


Figure 6.15. An example of two arguments for a call apply $(+, B_f, B_g)$.

of B_f and B_g downwards to construct nodes of the OBDD $B_{f \,\mathrm{op}\, g}$. Let r_f be the root node of B_f and r_g the root node of B_g.

1. If both r_f and r_g are terminal nodes with labels l_f and l_g, respectively (recall that terminal labels are either 0 or 1), then we compute the value $l_f \,\mathrm{op}\, l_g$ and let the resulting OBDD be B_0 if that value is 0 and B_1 otherwise.
2. In the remaining cases, at least one of the root nodes is a non-terminal. Suppose that both root nodes are x_i-nodes. Then we create an x_i-node n with a dashed line to apply $(\mathrm{op}, \mathrm{lo}(r_f), \mathrm{lo}(r_g))$ and a solid line to apply $(\mathrm{op}, \mathrm{hi}(r_f), \mathrm{hi}(r_g))$, i.e. we call apply recursively on the basis of (6.2).
3. If r_f is an x_i-node, but r_g is a terminal node or an x_j-node with $j > i$, then we know that there is no x_i-node in B_g because the two OBDDs have a compatible ordering of boolean variables. Thus, g is independent of x_i ($g \equiv g[0/x_i] \equiv g[1/x_i]$). Therefore, we create an x_i-node n with a dashed line to apply $(\mathrm{op}, \mathrm{lo}(r_f), r_g)$ and a solid line to apply $(\mathrm{op}, \mathrm{hi}(r_f), r_g)$.
4. The case in which r_g is a non-terminal, but r_f is a terminal or an x_j-node with $j > i$, is handled symmetrically to case 3.

The result of this procedure might not be reduced; therefore apply finishes by calling the function reduce on the OBDD it constructed. An example of apply (where op is $+$) can be seen in Figures 6.15–6.17. Figure 6.16 shows the recursive descent control structure of apply and Figure 6.17 shows the final result. In this example, the result of apply $(+, B_f, B_g)$ is B_f.

Figure 6.16 shows that numerous calls to apply occur several times with the same arguments. Efficiency could be gained if these were evaluated only

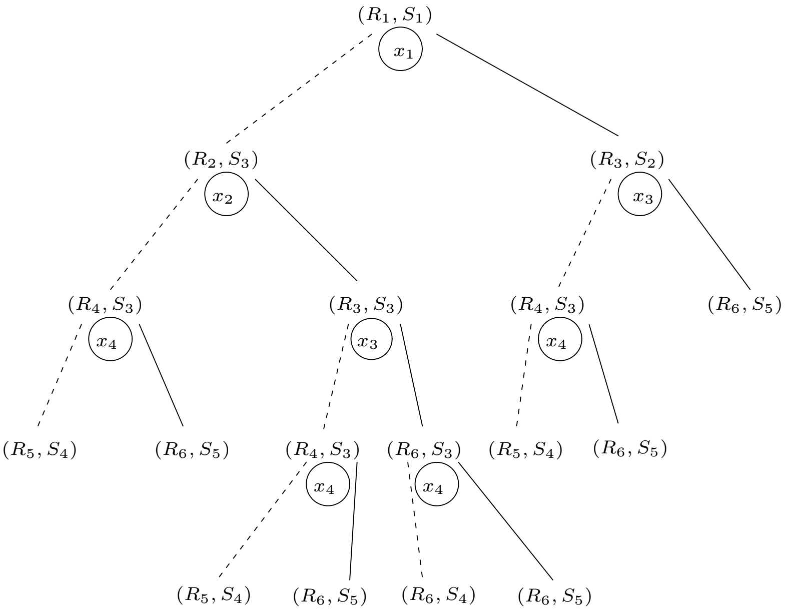


Figure 6.16. The recursive call structure of `apply` for the example in Figure 6.15 (without memoisation).

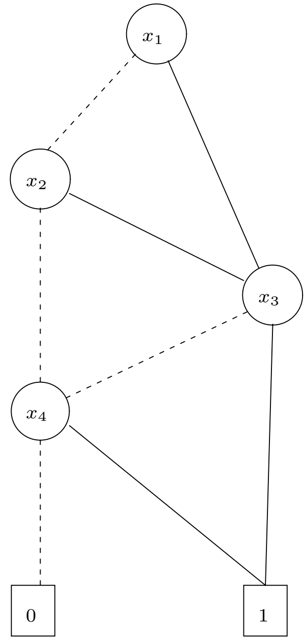


Figure 6.17. The result of `apply` $(+, B_f, B_g)$, where B_f and B_g are given in Figure 6.15.

the first time and the result remembered for future calls. This programming technique is known as memoisation. As well as being more efficient, it has the advantage that the resulting OBDD requires less reduction. (In this example, using memoisation eliminates the need for the final call to `reduce` altogether.) Without memoisation, `apply` is exponential in the size of its arguments, since each non-leaf call generates a further two calls. With memoisation, the number of calls to apply is bounded by $2 \cdot |B_f| \cdot |B_g|$, where $|B|$ is the size of the BDD. This is a worst-time complexity; the actual performance is often much better than this.

6.2.3 The algorithm `restrict`

Given an OBDD B_f representing a boolean formula f, we need an algorithm `restrict` such that the call `restrict`$(0, x, B_f)$ computes the reduced OBDD representing $f[0/x]$ using the same variable ordering as B_f. The algorithm for `restrict`$(0, x, B_f)$ works as follows. For each node n labelled with x, incoming edges are redirected to $\mathrm{lo}(n)$ and n is removed. Then we call `reduce` on the resulting OBDD. The call `restrict` $(1, x, B_f)$ proceeds similarly, only we now redirect incoming edges to $\mathrm{hi}(n)$.

6.2.4 The algorithm `exists`

A boolean function can be thought of as putting a constraint on the values of its argument variables. For example, the function $x + (\overline{y} \cdot z)$ evaluates to 1 only if x is 1; or y is 0 and z is 1 – this is a constraint on x, y, and z.

It is useful to be able to express the relaxation of the constraint on a subset of the variables concerned. To allow this, we write $\exists x.\, f$ for the boolean function f with the constraint on x relaxed. Formally, $\exists x.\, f$ is defined as $f[0/x] + f[1/x]$; that is, $\exists x.\, f$ is true if f could be made true by putting x to 0 or to 1. Given that $\exists x.\, f \stackrel{\text{def}}{=} f[0/x] + f[1/x]$ the `exists` algorithm can be implemented in terms of the algorithms `apply` and `restrict` as

$$\texttt{apply}\,(+, \texttt{restrict}\,(0, x, B_f), \texttt{restrict}\,(1, x, B_f))\ . \tag{6.3}$$

Consider, for example, the OBDD B_f for the function $f \stackrel{\text{def}}{=} x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$, shown in Figure 6.19. Figure 6.20 shows `restrict`$(0, x_3, B_f)$ and `restrict`$(1, x_3, B_f)$ and the result of applying $+$ to them. (In this case the apply function happens to return its second argument.)

We can improve the efficiency of this algorithm. Consider what happens during the `apply` stage of (6.3). In that case, the `apply` algorithm works on two BDDs which are identical all the way down to the level of the x-nodes;

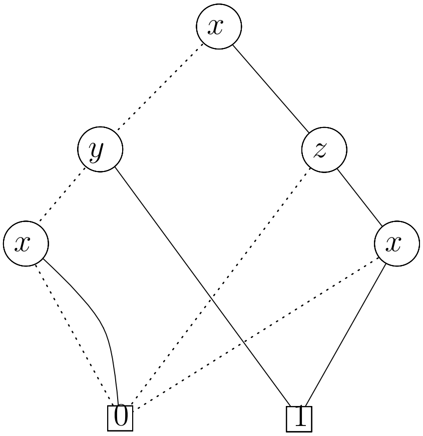


Figure 6.18. An example of a BDD which is not a read-1-BDD.

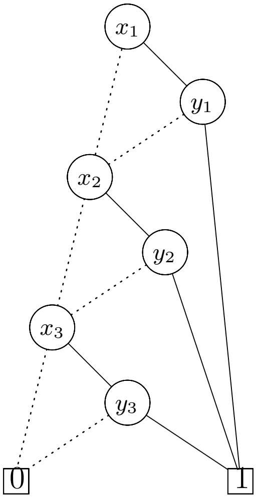


Figure 6.19. A BDD B_f to illustrate the `exists` algorithm.

therefore the returned BDD also has that structure down to the x-nodes. At the x-nodes, the two argument BDDs differ, so the `apply` algorithm will compute the apply of $+$ to these two subBDDs and return that as the subBDD of the result. This is illustrated in Figure 6.20. Therefore, we can compute the OBDD for $\exists x.\, f$ by taking the OBDD for f and replacing each node labelled with x by the result of calling `apply` on $+$ and its two branches.

This can easily be generalised to a sequence of `exists` operations. We write $\exists \hat{x}.\, f$ to mean $\exists x_1 . \exists x_2 . \ldots \exists x_n .\, f$, where $\hat{x}$ denotes $(x_1, x_2, \ldots, x_n)$.

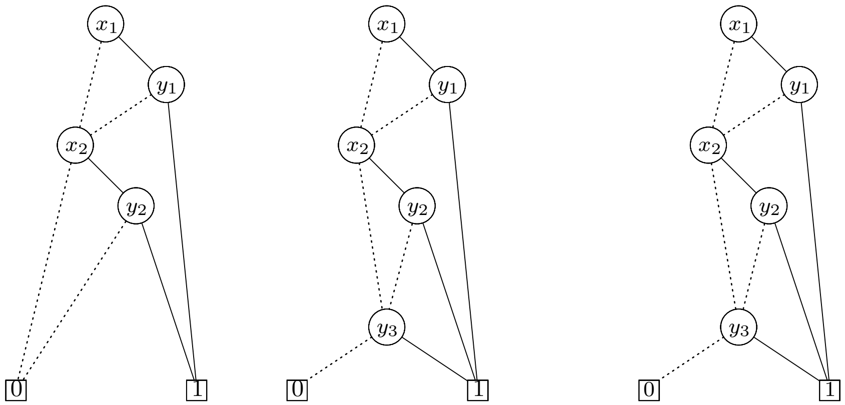


Figure 6.20. $\texttt{restrict}(0, x_3, B_f)$ and $\texttt{restrict}(1, x_3, B_f)$ and the result of applying $+$ to them.

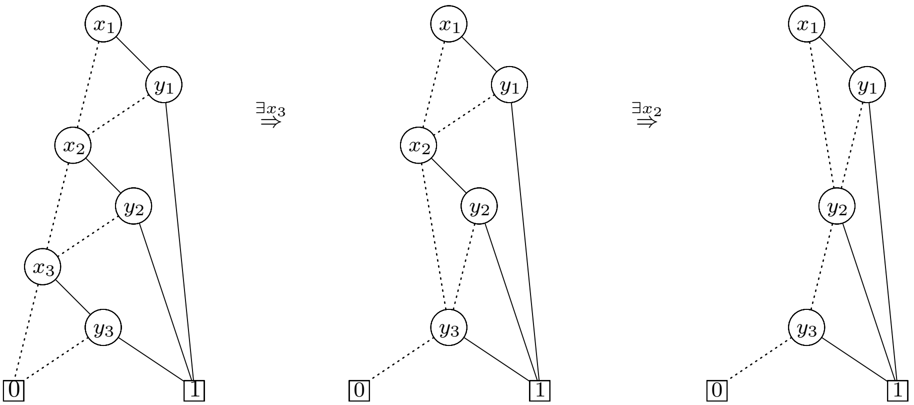

x_1
y_1
x_2
y_2
x_3
y_3
0
1
$\overset{\exists x_3}{\Rightarrow}$
$\overset{\exists x_2}{\Rightarrow}$

Figure 6.21. OBDDs for f, $\exists x_3. f$ and $\exists x_2.\exists x_3. f$.

The OBDD for this boolean function is obtained from the OBDD for f by replacing *every* node labelled with an x_i by the $+$ of its two branches.

Figure 6.21 shows the computation of $\exists x_3. f$ and $\exists x_2.\exists x_3. f$ (which is semantically equivalent to $x_1 \cdot y_1 + y_2 + y_3$) in this way.
The boolean quantifier $\forall$ is the dual of $\exists$:

$$\forall x.f \stackrel{\text{def}}{=} f[0/x] \cdot f[1/x]$$

asserting that f could be made false by putting x to 0 or to 1.

The translation of boolean formulas into OBDDs using the algorithms of this section is summarised in Figure 6.22.

Boolean formula f	Representing OBDD B_f
0	B_0 (Fig. 6.6)
1	B_1 (Fig. 6.6)
x	B_x (Fig. 6.6)
$\overline{f}$	swap the 0- and 1-nodes in B_f
$f + g$	`apply` $(+, B_f, B_g)$
$f \cdot g$	`apply` $(\cdot\,, B_f, B_g)$
$f \oplus g$	`apply` $(\oplus, B_f, B_g)$
$f[1/x]$	`restrict` $(1, x, B_f)$
$f[0/x]$	`restrict` $(0, x, B_f)$
$\exists x.f$	`apply` $(+, B_{f[0/x]}, B_{f[1/x]})$
$\forall x.f$	`apply` $(\cdot\,, B_{f[0/x]}, B_{f[1/x]})$

Figure 6.22. Translating boolean formulas f to OBDDs B_f, given a fixed, global ordering on boolean variables.

Algorithm	Input OBDD(s)	Output OBDD	Time-complexity
`reduce`	B	reduced B	$O(\lvert B\rvert \cdot \log \lvert B\rvert)$
`apply`	B_f, B_g (reduced)	$B_{f \text{ op } g}$ (reduced)	$O(\lvert B_f\rvert \cdot \lvert B_g\rvert)$
`restrict`	B_f (reduced)	$B_{f[0/x]}$ or $B_{f[1/x]}$ (reduced)	$O(\lvert B_f\rvert \cdot \log \lvert B_f\rvert)$
$\exists$	B_f (reduced)	$B_{\exists x_1.\exists x_2.\dots\exists x_n.f}$ (reduced)	NP-complete

Figure 6.23. Upper bounds in terms of the input OBDD(s) for the worst-case running times of our algorithms needed in our implementation of boolean formulas.

6.2.5 Assessment of OBDDs

Time complexities for computing OBDDs We can measure the complexity of the algorithms of the preceding section by giving upper bounds for the running time in terms of the sizes of the input OBDDs. The table in Figure 6.23 summarises these upper bounds (some of those upper bounds may require more sophisticated versions of the algorithms than the versions presented in this chapter). All the operations except nested boolean quantification are practically efficient in the size of the participating OBDDs. Thus, modelling very large systems with this approach will work if the OBDDs which represent the systems don't grow too large too fast. If we can somehow control the size of OBDDs, e.g. by using good heuristics for the choice of variable ordering, then these operations are computationally feasible. It has already been shown that OBDDs modelling certain classes of systems and networks don't grow excessively.

The expensive computational operations are the nested boolean quantifications $\exists z_1 .\ldots\exists z_n . f$ and $\forall z_1 .\ldots\forall z_n . f$. By exercise 1 on page 406, the computation of the OBDD for $\exists z_1 .\ldots\exists z_n . f$, given the OBDD for f, is an NP-complete problem[2]; thus, it is unlikely that there exists an algorithm with a feasible worst-time complexity. This is not to say that boolean functions modelling practical systems may not have efficient nested boolean quantifications. The performance of our algorithms can be improved by using further optimisation techniques, such as parallelisation.

Note that the operations `apply`, `restrict`, etc. are only efficient in the size of the input OBDDs. So if a function f does not have a compact representation as an OBDD, then computing with its OBDD will not be efficient. There are such nasty functions; indeed, one of them is *integer multiplication*. Let $b_{n-1}b_{n-2}\ldots b_0$ and $a_{n-1}a_{n-2}\ldots a_0$ be two n-bit integers, where b_{n-1} and a_{n-1} are the most significant bits and b_0 and a_0 are the least significant bits. The multiplication of these two integers results in a $2n$-bit integer. Thus, we may think of multiplication as $2n$ many boolean functions f_i in $2n$ variables (n bits for input b and n bits for input a), where f_i denotes the ith output bit of the multiplication. The following negative result, due to R. E. Bryant, shows that OBDDs cannot be used for implementing integer multiplication.

Theorem 6.11 Any OBDD representation of f_{n-1} has at least a number of vertices proportional to 1.09^n, i.e. its size is exponential in n.

Extensions and variations of OBDDs There are many variations and extensions to the OBDD data structure. Many of them can implement certain operations more efficiently than their OBDD counterparts, but it seems that none of them perform as well as OBDDs overall. In particular, one feature which many of the variations lack is the canonical form; therefore they lack an efficient algorithm for deciding when two objects denote the same boolean function.

One kind of variation allows non-terminal nodes to be labelled with binary operators as well as boolean variables. *Parity OBDDs* are like OBDDs in that there is an ordering on variables and every variable may occur at

[2] Another NP-complete problem is to decide the satisfiability of formulas of propositional logic.

most once on a path; but some non-terminal nodes may be labelled with $\oplus$, the exclusive-or operation. The meaning is that the function represented by that node is the exclusive-or of the boolean functions determined by its children. Parity OBDDs have similar algorithms for `apply`, `restrict`, etc. with the same performance, but they do not have a canonical form. Checking for equivalence cannot be done in constant time. There is, however, a cubic algorithm for determining equivalence; and there are also efficient probabilistic tests. Another variation of OBDDs allows complementation nodes, with the obvious meaning. Again, the main disadvantage is the lack of canonical form.

One can also allow non-terminal nodes to be unlabelled and to branch to more than two children. This can then be understood either as nondeterministic branching, or as probabilistic branching: throw a pair of dice to determine where to continue the path. Such methods may compute wrong results; one then aims at repeating the test to keep the (probabilistic) error as small as desired. This method of repeating probabilistic tests is called *probabilistic amplification*. Unfortunately, the satisfiability problem for probabilistic branching OBDDs is NP-complete. On a good note, probabilistic branching OBDDs can *verify* integer multiplication.

The development of extensions or variations of OBDDS which are customised to certain classes of boolean functions is an important area of ongoing research.

6.3 Symbolic model checking

The use of BDDs in model checking resulted in a significant breakthrough in verification in the early 1990s, because they have allowed systems with much larger state spaces to be verified. In this section, we describe in detail how the model-checking algorithm presented in Chapter 3 can be implemented using OBDDs as the basic data structure.

The pseudo-code presented in Figure 3.28 on page 227 takes as input a CTL formula ϕ and returns the set of states of the given model which satisfy ϕ. Inspection of the code shows that the algorithm consists of manipulating intermediate sets of states. We show in this section how the model and the intermediate sets of states can be stored as OBDDs; and how the operations required in that pseudo-code can be implemented in terms of the operations on OBDDs which we have seen in this chapter.

We start by showing how sets of states are represented with OBDDs, together with some of the operations required. Then, we extend that to the representation of the transition system; and finally, we show how the remainder of the required operations is implemented.

Model checking using OBDDs is called *symbolic model checking*. The term emphasises that individual states are not represented; rather, sets of states are represented symbolically, namely, those which satisfy the formula being checked.

6.3.1 Representing subsets of the set of states

Let S be a finite set (we forget for the moment that it is a set *of states*). The task is to represent the various subsets of S as OBDDs. Since OBDDs encode boolean functions, we need somehow to code the elements of S as boolean values. The way to do this in general is to assign to each element $s \in S$ a unique vector of boolean values $(v_1, v_2, \dots, v_n)$, each $v_i \in \{0, 1\}$. Then, we represent a subset T by the boolean function f_T which maps $(v_1, v_2, \dots, v_n)$ onto 1 if $s \in T$ and maps it onto 0 otherwise.

There are 2^n boolean vectors $(v_1, v_2, \dots, v_n)$ of length n. Therefore, n should be chosen such that $2^{n-1} < |S| \leq 2^n$, where $|S|$ is the number of elements in S. If $|S|$ is not an exact power of 2, there will be some vectors which do not correspond to any element of S; they are just ignored. The function $f_T : \{0, 1\}^n \to \{0, 1\}$ which tells us, for each s, represented by $(v_1, v_2, \dots, v_n)$, whether it is in the set T or not, is called the *characteristic function* of T.

In the case that S is the set of states of a transition system $\mathcal{M} = (S, \to, L)$ (see Definition 3.4), there is a natural way of choosing the representation of S as boolean vectors. The labelling function $L : S \to \mathcal{P}(\mathtt{Atoms})$ (where $\mathcal{P}(\mathtt{Atoms})$ is the set of subsets of `Atoms`) gives us the encoding. We assume a fixed ordering on the set `Atoms`, say $x_1, x_2, \dots, x_n$, and then represent $s \in S$ by the vector $(v_1, v_2, \dots, v_n)$, where, for each i, v_i equals 1 if $x_i \in L(s)$ and v_i is 0 otherwise. In order to guarantee that each s has a unique representation as a boolean vector, we require that, for all $s_1, s_2 \in S$, $L(s_1) = L(s_2)$ implies $s_1 = s_2$. If this is not the case, perhaps because $2^{|\mathtt{Atoms}|} < |S|$, we can add extra atomic propositions in order to make enough distinctions (*Cf.* introduction of the `turn` variable for mutual exclusion in Section 3.3.4.)

From now on, we refer to a state $s \in S$ by its representing boolean vector $(v_1, v_2, \dots, v_n)$, where v_i is 1 if $x_i \in L(s)$ and 0 otherwise. As an OBDD, this state is represented by the OBDD of the boolean function $l_1 \cdot l_2 \cdot \dots \cdot l_n$, where l_i is x_i if $x_i \in L(s)$ and $\overline{x_i}$ otherwise. The set of states $\{s_1, s_2, \dots, s_m\}$ is represented by the OBDD of the boolean function

$$(l_{11} \cdot l_{12} \cdot \dots \cdot l_{1n}) + (l_{21} \cdot l_{22} \cdot \dots \cdot l_{2n}) + \dots + (l_{m1} \cdot l_{m2} \cdot \dots \cdot l_{mn})$$

where $l_{i1} \cdot l_{i2} \cdot \dots \cdot l_{in}$ represents state s_i.

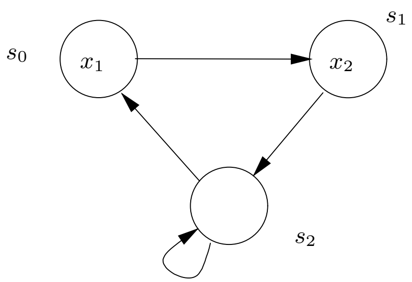


Figure 6.24. A simple CTL model (Example 6.12).

set of states	representation by boolean values	representation by boolean function
$\emptyset$		0
$\{s_0\}$	$(1,0)$	$x_1 \cdot \overline{x_2}$
$\{s_1\}$	$(0,1)$	$\overline{x_1} \cdot x_2$
$\{s_2\}$	$(0,0)$	$\overline{x_1} \cdot \overline{x_2}$
$\{s_0, s_1\}$	$(1,0)$, $(0,1)$	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$
$\{s_0, s_2\}$	$(1,0)$, $(0,0)$	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_2}$
$\{s_1, s_2\}$	$(0,1)$, $(0,0)$	$\overline{x_1} \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$
S	$(1,0)$, $(0,1)$, $(0,0)$	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$

Figure 6.25. Representation of subsets of states of the model of Figure 6.24.

The key point which makes this representation interesting is that the OBDD representing a set of states may be quite small.

Example 6.12 Consider the CTL model in Figure 6.24, given by:

$$\begin{aligned} S &\stackrel{\text{def}}{=} \{s_0, s_1, s_2\} \\ \to &\stackrel{\text{def}}{=} \{(s_0, s_1),\ (s_1, s_2),\ (s_2, s_0),\ (s_2, s_2)\} \\ L(s_0) &\stackrel{\text{def}}{=} \{x_1\} \\ L(s_1) &\stackrel{\text{def}}{=} \{x_2\} \\ L(s_2) &\stackrel{\text{def}}{=} \emptyset. \end{aligned}$$

Note that it has the property that, for all states s_1 and s_2, $L(s_1) = L(s_2)$ implies $s_1 = s_2$, i.e. a state is determined entirely by the atomic formulas true in it. Sets of states may be represented by boolean values and by boolean formulas with the ordering $[x_1, x_2]$, as shown in Figure 6.25.

Notice that the vector $(1,1)$ and the corresponding function $x_1 \cdot x_2$ are unused. Therefore, we are free to include it in the representation of a subset

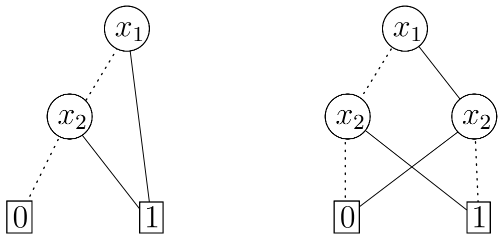


Figure 6.26. Two OBDDs for the set $\{s_0, s_1\}$ (Example 6.12).

of S or not; so we may choose to include it or not in order to optimise the size of the OBDD. For example, the subset $\{s_0, s_1\}$ is better represented by the boolean function $x_1 + x_2$, since its OBDD is smaller than that for $x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$ (Figure 6.26).

In order to justify the claim that the representation of subsets of S as OBDDs will be suitable for the algorithm presented in Section 3.6.1, we need to look at how the operations on subsets which are used in that algorithm can be implemented in terms of the operations we have defined on OBDDs. The operations in that algorithm are:

- Intersection, union and complementation of subsets. It is clear that these are represented by the boolean functions $\cdot$, $+$ and $\bar{\ }$ respectively. The implementation via OBDDs of $\cdot$ and $+$ uses the `apply` algorithm (Section 6.2.2).
- The functions

$$\begin{aligned} \mathrm{pre}_\exists(X) &= \{s \in S \mid \text{exists } s',\ (s \to s' \text{ and } s' \in X)\} \\ \mathrm{pre}_\forall(X) &= \{s \mid \text{for all } s',\ (s \to s' \text{ implies } s' \in X)\}. \end{aligned} \tag{6.4}$$

  The function $\mathrm{pre}_\exists$ (instrumental in $\mathtt{SAT}_{\mathtt{EX}}$ and $\mathtt{SAT}_{\mathtt{EU}}$) takes a subset X of states and returns the set of states which can make a transition into X. The function $\mathrm{pre}_\forall$, used in $\mathtt{SAT}_{\mathtt{AF}}$, takes a set X and returns the set of states which can make a transition *only* into X. In order to see how these are implemented in terms of OBDDs, we need first to look at how the transition relation itself is represented.

6.3.2 Representing the transition relation

The transition relation $\to$ of a model $\mathcal{M} = (S, \to, L)$ is a subset of $S \times S$. We have already seen that subsets of a given finite set may be represented as OBDDs by considering the characteristic function of a binary encoding.

Just like in the case of subsets of S, the binary encoding is naturally given by the labelling function L. Since $\to$ is a subset of $S \times S$, we need two copies of the boolean vectors. Thus, the link $s \to s'$ is represented by the pair of

x_1	x_2	x'_1	x'_2	$\rightarrow$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

x_1	x'_1	x_2	x'_2	$\rightarrow$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 6.27. The truth table for the transition relation of Figure 6.24 (see Example 6.13). The left version shows the ordering of variables $[x_1, x_2, x'_1, x'_2]$, while the right one orders the variables $[x_1, x'_1, x_2, x'_2]$ (the rows are ordered lexicographically).

boolean vectors $((v_1, v_2, \ldots, v_n), (v'_1, v'_2, \ldots, v'_n))$, where v_i is 1 if $p_i \in L(s)$ and 0 otherwise; and similarly, v'_i is 1 if $p_i \in L(s')$ and 0 otherwise. As an OBDD, the link is represented by the OBDD for the boolean function

$$(l_1 \cdot l_2 \cdot \cdots \cdot l_n) \cdot (l'_1 \cdot l'_2 \cdot \cdots \cdot l'_n)$$

and a set of links (for example, the entire relation →) is the OBDD for the $+$ of such formulas.

Example 6.13 To compute the OBDD for the transition relation of Figure 6.24, we first show it as a truth table (Figure 6.27 (left)). Each 1 in the final column corresponds to a link in the transition relation and each 0 corresponds to the absence of a link. The boolean function is obtained by taking the disjunction of the rows having 1 in the last column and is

$$f^{\rightarrow} \overset{\mathrm{def}}{=} \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}'_1 \cdot \overline{x}'_2 + \overline{x}_1 \cdot \overline{x}_2 \cdot x'_1 \cdot \overline{x}'_2 + x_1 \cdot \overline{x}_2 \cdot \overline{x}'_1 \cdot x'_2 + \overline{x}_1 \cdot x_2 \cdot \overline{x}'_1 \cdot \overline{x}'_2. \tag{6.5}$$

It turns out that it is usually more efficient to interleave unprimed and primed variables in the OBDD variable ordering for →. We therefore use

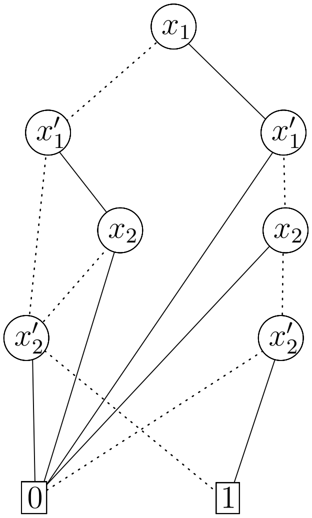


Figure 6.28. An OBDD for the transition relation of Example 6.13.

$[x_1, x_1', x_2, x_2']$ rather than $[x_1, x_2, x_1', x_2']$. Figure 6.27 (right) shows the truth table redrawn with the interleaved ordering of the columns and the rows reordered lexicographically. The resulting OBDD is shown in Figure 6.28.

6.3.3 Implementing the functions $\text{pre}_\exists$ and $\text{pre}_\forall$

It remains to show how an OBDD for $\text{pre}_\exists(X)$ and $\text{pre}_\forall(X)$ can be computed, given OBDDs B_X for X and $B_\to$ for the transition relation $\to$. First we observe that $\text{pre}_\forall$ can be expressed in terms of complementation and $\text{pre}_\exists$, as follows: $\text{pre}_\forall(X) = S - \text{pre}_\exists(S - X)$, where we write $S - Y$ for the set of all $s \in S$ which are not in Y. Therefore, we need only explain how to compute the OBDD for $\text{pre}_\exists(X)$ in terms of B_X and $B_\to$. Now (6.4) suggests that one should proceed as follows:

1. Rename the variables in B_X to their primed versions; call the resulting OBDD $B_{X'}$.
2. Compute the OBDD for $\texttt{exists}(\hat{x}', \texttt{apply}(\cdot, B_\to, B_{X'}))$ using the `apply` and `exists` algorithms (Sections 6.2.2 and 6.2.4).

6.3.4 Synthesising OBDDs

The method used in Example 6.13 for producing an OBDD for the transition relation was to compute first the truth table and then an OBDD which might not be in its fully reduced form; hence the need for a final call to the `reduce` function. However, this procedure would be unacceptable if applied to realistically sized systems with a large number of variables, for the truth table's size is exponential in the number of boolean variables. The key idea and attraction of applying OBDDs to finite systems is therefore to take a system description in a language such as SMV and to synthesise the OBDD directly, without having to go via intermediate representations (such as binary decision *trees* or truth tables) which are exponential in size.

SMV allows us to define the next value of a variable in terms of the current values of variables (see the examples of code in Section 3.3.2)[3]. This can be compiled into a set of boolean functions f_i, one for each variable x_i, which define the next value of x_i in terms of the current values of all the variables. In order to cope with non-deterministic assignment (such as the assignment to `status` in the example on page 192), we extend the set of variables by adding unconstrained variables which model the input. Each x'_i is a deterministic function of this enlarged set of variables; thus, $x'_i \leftrightarrow f_i$, where $f \leftrightarrow g = 1$ if, and only if, f and g compute the same values, i.e. it is a shorthand for $\overline{f \oplus g}$.

The boolean function representing the transition relation is therefore of the form

$$\prod_{1 \leq i \leq n} x'_i \leftrightarrow f_i, \tag{6.6}$$

where $\prod_{1 \leq i \leq n} g_i$ is a shorthand for $g_1 \cdot g_2 \cdot \ldots \cdot g_n$. Note that the $\prod$ ranges only over the non-input variables. So, if u is an input variable, the boolean function does not contain any $u' \leftrightarrow f_u$.

Figure 6.22 showed how the reduced OBDD could be computed from the parse tree of such a boolean function. Thus, it is possible to compile SMV programs into OBDDs such that their specifications can be executed according to the pseudo-code of the function `SAT`, now interpreted over OBDDs. On page 396 we will see that this OBDD implementation can be extended to simple fairness constraints.

Modelling sequential circuits As a further application of OBDDs to verification, we show how OBDDs representing circuits may be synthesised.

Synchronous circuits. Suppose that we have a design of a sequential circuit such as the one in Figure 6.29. This is a synchronous circuit (meaning that

[3] SMV also allows next values to be defined in terms of next values, i.e. the keyword `next` to appear in expressions on the right-hand side of :=. This is useful for describing synchronisations, for example, but we ignore that feature here.

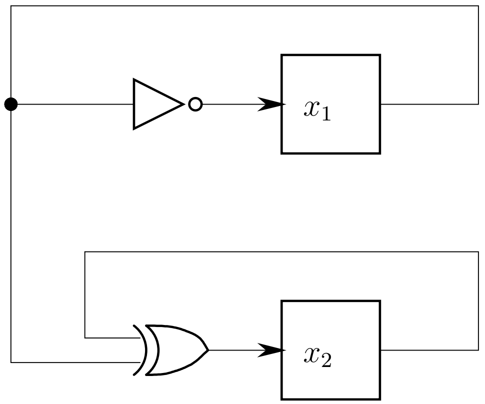


Figure 6.29. A simple synchronous circuit with two registers.

all the state variables are updated synchronously in parallel) whose functionality can be described by saying what the values of the registers x_1 and x_2 in the next state of the circuit are. The function $f^{\rightarrow}$ coding the possible next states of the circuits is

$$(x_1' \leftrightarrow \overline{x}_1) \cdot (x_2' \leftrightarrow x_1 \oplus x_2). \tag{6.7}$$

This may now be translated into an OBDD by the methods summarised in Figure 6.22.

Asynchronous circuits. The symbolic encoding of synchronous circuits is in its logical structure very similar to the encoding of $f^{\rightarrow}$ for CTL models; compare the codings in (6.7) and (6.6). In asynchronous circuits, or processes in SMV, the logical structure of $f^{\rightarrow}$ changes. As before, we can construct functions f_i which code the possible next state in the *local component*, or the SMV process, i. For asynchronous systems, there are two principal ways of composing these functions into global system behaviour:

- In a *simultaneous model*, a global transition is one in which any number of components may make their local transition. This is modelled as

$$f^{\rightarrow} \stackrel{\text{def}}{=} \prod_{i=1}^{n} \left((x_i' \leftrightarrow f_i) + (x_i' \leftrightarrow x_i)\right). \tag{6.8}$$

- In an *interleaving model*, exactly one local component makes a local transition;

all other local components remain in their local state:

$$f^{\rightarrow} \stackrel{\text{def}}{=} \sum_{i=1}^{n} \left((x'_i \leftrightarrow f_i) \cdot \prod_{j \neq i} (x'_j \leftrightarrow x_j) \right). \tag{6.9}$$

Observe the duality in these approaches: the simultaneous model has an outer product, whereas the interleaving model has an outer sum. The latter, if used in $\exists \hat{x}'.f$ ('for some next state'), can be optimised since sums distribute over existential quantification; in Chapter 2 this was the equivalence $\exists x.(\phi \vee \psi) \equiv \exists x.\phi \vee \exists x.\psi$. Thus, global states reachable in one step are the 'union' of all the states reachable in one step in the local components; compare the formulas in (6.8) and (6.9) with (6.6).

6.4 A relational mu-calculus

We saw in Section 3.7 that evaluating the set of states satisfying a CTL formula in a model may involve the computation of a fixed point of an operator. For example, $[\![\mathrm{EF}\,\phi]\!]$ is the least fixed point of the operator $F \colon \mathcal{P}(S) \to \mathcal{P}(S)$ given by $F(X) = [\![\phi]\!] \cup \mathrm{pre}_{\exists}(X)$.

In this section, we introduce a syntax for referring to fixed points in the context of boolean formulas. Fixed-point invariants frequently occur in all sorts of applications (for example, the common-knowledge operator C_G in Chapter 5), so it makes sense to have an intermediate language for expressing such invariants syntactically. This language also provides a formalism for describing interactions and dependences of such invariants. We will see shortly that symbolic model checking in the presence of simple fairness constraints exhibits such more complex relationships between invariants.

6.4.1 Syntax and semantics

Definition 6.14 The formulas of the relational mu-calculus are given by the grammar

$$\begin{aligned} v &::= x \mid Z \\ f &::= 0 \mid 1 \mid v \mid \overline{f} \mid f_1 + f_2 \mid f_1 \cdot f_2 \mid f_1 \oplus f_2 \mid \\ &\qquad \exists x.f \mid \forall x.f \mid \mu Z.f \mid \nu Z.f \mid f[\hat{x} := \hat{x}'] \end{aligned} \tag{6.10}$$

where x and Z are boolean variables, and $\hat{x}$ is a tuple of variables. In the formulas $\mu Z.f$ and $\nu Z.f$, any occurrence of Z in f is required to fall within an even number of complementation symbols $\bar{\ }$; such an f is said to be formally monotone in Z. (In exercise 7 on page 410 we consider what happens if we do not require formal monotonicity.)

Convention 6.15 The binding priorities for the grammar in (6.10) are that $\bar{}$, and $[\hat{x} := \hat{x}']$ have the highest priority; followed by $\exists x$ and $\forall y$; then μZ and νZ; followed by $\cdot$. The operators $+$ and $\oplus$ have the lowest binding priority.

The symbols μ and ν are called *least fixed-point* and *greatest fixed-point* operators, respectively. In the formula $\mu Z.f$, the interesting case is that in which f contains an occurrence of Z. In that case, f can be thought of as a function, taking Z to f. The formula $\mu Z.f$ is intended to mean the least fixed point of that function. Similarly, $\nu Z.f$ is the greatest fixed point of the function. We will see how this is done in the semantics.

The formula $f[\hat{x} := \hat{x}']$ expresses an explicit substitution which forces f to be evaluated using the values of x_i' rather than x_i. (Recall that the primed variables refer to the next state.) Thus, this syntactic form is not a meta-operation denoting a substitution, but an explicit syntactic form in its own right. The substitution will be made on the semantic side, not the syntactic side. This difference will become clear when we present the semantics of $\vDash$.

A valuation ρ for f is an assignment of values 0 or 1 to all variables v. We define a *satisfaction relation* $\rho \vDash f$ inductively over the structure of such formulas f, given a valuation ρ.

Definition 6.16 Let ρ be a valuation and v a variable. We write $\rho(v)$ for the value of v assigned by ρ. We define $\rho[v \mapsto 0]$ to be the updated valuation which assigns 0 to v and $\rho(w)$ to all other variables w. Dually, $\rho[v \mapsto 1]$ assigns 1 to v and $\rho(w)$ to all other variables w.

For example, if ρ is the valuation represented by $(x, y, Z) \Rightarrow (1, 0, 1)$ – meaning that $\rho(x) = 1$, $\rho(y) = 0$, $\rho(Z) = 1$ and $\rho(v) = 0$ for all other variables v – then $\rho[x \mapsto 0]$ is represented by $(x, y, Z) \Rightarrow (0, 0, 1)$, whereas $\rho[Z \mapsto 0]$ is $(x, y, Z) \Rightarrow (1, 0, 0)$. The assumption that valuations assign values to all variables is rather mathematical, but avoids some complications which have to be addressed in implementations (see exercise 3 on page 409).

Updated valuations allow us to define the satisfaction relation for all formulas without fixed points:

Definition 6.17 We define a satisfaction relation $\rho \vDash f$ for formulas f without fixed-point subformulas with respect to a valuation ρ by structural induction:

- $\rho \nvDash 0$
- $\rho \vDash 1$
- $\rho \vDash v$ iff $\rho(v)$ equals 1

- $\rho \vDash \overline{f}$ iff $\rho \nvDash f$
- $\rho \vDash f + g$ iff $\rho \vDash f$ or $\rho \vDash g$
- $\rho \vDash f \cdot g$ iff $\rho \vDash f$ and $\rho \vDash g$
- $\rho \vDash f \oplus g$ iff $\rho \vDash (f \cdot \overline{g} + \overline{f} \cdot g)$
- $\rho \vDash \exists x.f$ iff $\rho[x \mapsto 0] \vDash f$ or $\rho[x \mapsto 1] \vDash f$
- $\rho \vDash \forall x.f$ iff $\rho[x \mapsto 0] \vDash f$ and $\rho[x \mapsto 1] \vDash f$
- $\rho \vDash f[\hat{x} := \hat{x}']$ iff $\rho[\hat{x} := \hat{x}'] \vDash f$,

where $\rho[\hat{x} := \hat{x}']$ is the valuation which assigns the same values as ρ, but for each x_i it assigns $\rho(x_i')$.

The semantics of boolean quantification closely resembles the one for the quantifiers of predicate logic. The crucial difference, however, is that boolean formulas are only interpreted over the fixed universe of values $\{0, 1\}$, whereas predicate formulas may take on values in all sorts of finite or infinite models.

Example 6.18 Let ρ be such that $\rho(x_1')$ equals 0 and $\rho(x_2')$ is 1. We evaluate $\rho \vDash (x_1 + \overline{x}_2)[\hat{x} := \hat{x}']$ which holds iff $\rho[\hat{x} := \hat{x}'] \vDash (x_1 + \overline{x}_2)$. Thus, we need $\rho[\hat{x} := \hat{x}'] \vDash x_1$ or $\rho[\hat{x} := \hat{x}'] \vDash \overline{x}_2$ to be the case. Now, $\rho[\hat{x} := \hat{x}'] \vDash x_1$ cannot be, for this would mean that $\rho(x_1')$ equals 1. Since $\rho[\hat{x} := \hat{x}'] \vDash \overline{x}_2$ would imply that $\rho[\hat{x} := \hat{x}'] \nvDash x_2$, we infer that $\rho[\hat{x} := \hat{x}'] \nvDash \overline{x}_2$ because $\rho(x_2')$ equals 1. In summary, we demonstrated that $\rho \nvDash (x_1 + \overline{x}_2)[\hat{x} := \hat{x}']$.

We now extend the definition of $\vDash$ to the fixed-point operators μ and ν. Their semantics will have to reflect their meaning as least, respectively greatest, fixed-point operators. We define the semantics of $\mu Z.f$ via its syntactic approximants which unfold the meaning of $\mu Z.f$:

$$\begin{aligned} \mu_0 Z.f &\overset{\text{def}}{=} 0 \\ \mu_{m+1} Z.f &\overset{\text{def}}{=} f[\mu_m Z.f/Z] \qquad (m \geq 0). \end{aligned} \tag{6.11}$$

The unfolding is achieved by a meta-operation $[g/Z]$ which, when applied to a formula f, replaces all free occurrences of Z in f with g. Thus, we view μZ as a binding construct similar to the quantifiers $\forall x$ and $\exists x$, and $[g/Z]$ is similar to the substitution $[t/x]$ in predicate logic. For example, $(x_1 + \exists x_2.(Z \cdot x_2))[\overline{x}_1/Z]$ is the formula $x_1 + \exists x_2.(\overline{x}_1 \cdot x_2)$, whereas $((\mu Z.x_1 + Z) \cdot (x_1 + \exists x_2.(Z \cdot x_2)))[\overline{x}_1/Z]$ equals $(\mu Z.x_1 + Z) \cdot (x_1 + \exists x_2.(\overline{x}_1 \cdot x_2))$. See exercise 3 on page 409 for a formal account of this meta-operation.

With these approximants we can define:

$$\rho \vDash \mu Z.f \text{ iff } (\rho \vDash \mu_m Z.f \text{ for some } m \geq 0). \tag{6.12}$$

Thus, to determine whether $\mu Z.f$ is true with respect to a valuation ρ, we have to find some $m \geq 0$ such that $\rho \vDash \mu_m Z.f$ holds. A sensible strategy is to try to prove this for the smallest such m possible, if indeed such an m can be found. For example, in attempting to show $\rho \vDash \mu Z.Z$, we try $\rho \vDash \mu_0 Z.Z$, which fails since the latter formula is just 0. Now, $\mu_1 Z.Z$ is defined to be $Z[\mu_0 Z.Z/Z]$ which is just $\mu_0 Z.Z$ again. We can now use mathematical induction on $m \geq 0$ to show that $\mu_m Z.Z$ equals $\mu_0 Z.Z$ for all $m \geq 0$. By (6.12), this implies $\rho \nvDash \mu Z.Z$.

The semantics for $\nu Z.f$ is similar. First, let us define a family of approximants $\nu_0 Z.f$, $\nu_1 Z.f$, ... by

$$\begin{aligned} \nu_0 Z.f &\overset{\text{def}}{=} 1 \\ \nu_{m+1} Z.f &\overset{\text{def}}{=} f[\nu_m Z.f/Z] \qquad (m \geq 0). \end{aligned} \tag{6.13}$$

Note that this definition only differs from the one for $\mu_m Z.f$ in that the first approximant is defined to be 1 instead of 0.

Recall how the greatest fixed point for EG ϕ requires that ϕ holds on all states of some path. Such invariant behaviour cannot be expressed with a condition such as in (6.12), but is adequately defined by demanding that

$$\rho \vDash \nu Z.f \text{ iff } (\rho \vDash \nu_m Z.f \text{ for all } m \geq 0). \tag{6.14}$$

A dual reasoning to the above shows that $\rho \vDash \nu Z.Z$ holds, regardless of the nature of ρ.

One informal way of understanding the definitions in (6.12) and (6.14) is that $\rho \vDash \mu Z.f$ is false until, and if, it is proven to hold; whereas $\rho \vDash \nu Z.f$ is true until, and if, it is proven to be false. The temporal aspect is encoded by the unfolding of the recursion in (6.11), or in (6.13).

To prove that this recursive way of specifying $\rho \vDash f$ actually is well defined, one has to consider more general forms of induction which keep track not only of the height of f's parse tree, but also of the number of syntactic approximants $\mu_m Z.g$ and $\nu_n Z.h$, their 'degree' (in this case, m and n), as well as their 'alternation' (the body of a fixed point may contain a free occurrence of a variable for a recursion higher up in the parse tree). This can be done, though we won't discuss the details here.

6.4.2 Coding CTL models and specifications

Given a CTL model $\mathcal{M} = (S, \rightarrow, L)$, the μ and ν operators permit us to translate any CTL formula ϕ into a formula, f^ϕ, of the relational mu-calculus such that f^ϕ represents the set of states $s \in S$ with $s \vDash \phi$. Since we already saw how to represent subsets of states as such formulas, we can then capture the model-checking problem

$$\mathcal{M}, I \overset{?}{\vDash} \phi \tag{6.15}$$

of whether all *initial* states $s \in I$ satisfy ϕ, in purely symbolic form: we answer in the affirmative if $f^I \cdot \overline{f}^{\phi}$ is unsatisfiable, where f^I is the characteristic function of $I \subseteq S$. Otherwise, the logical structure of $f^I \cdot \overline{f}^{\phi}$ may be exploited to extract debugging information for correcting the model $\mathcal{M}$ in order to make (6.15) true.

Recall how we can represent the transition relation $\rightarrow$ as a boolean formula $f^{\rightarrow}$ (see Section 6.3.2). As before, we assume that states are coded as bit vectors $(v_1, v_2, \dots, v_n)$ and so the free boolean variables of all functions f^{ϕ} are subsumed by the vector $\hat{x}$. The coding of the CTL formula ϕ as a function f^{ϕ} in the relational mu-calculus is now given inductively as follows:

$$\begin{aligned} f^{x} &\overset{\text{def}}{=} x \quad \text{for variables } x \\ f^{\perp} &\overset{\text{def}}{=} 0 \\ f^{\neg\phi} &\overset{\text{def}}{=} \overline{f^{\phi}} \\ f^{\phi\wedge\psi} &\overset{\text{def}}{=} f^{\phi} \cdot f^{\psi} \\ f^{\mathrm{EX}\,\phi} &\overset{\text{def}}{=} \exists \hat{x}'.\, (f^{\rightarrow} \cdot f^{\phi}[\hat{x} := \hat{x}']). \end{aligned}$$

The clause for EX deserves explanation. The variables x_i refer to the current state, whereas x_i' refer to the next state. The semantics of CTL says that $s \vDash \mathrm{EX}\,\phi$ if, and only if, there is some s' with $s \rightarrow s'$ and $s' \vDash \phi$. The boolean formula encodes this definition, computing 1 precisely when this is the case. If $\hat{x}$ models the current state s, then $\hat{x}'$ models a possible successor state if $f^{\rightarrow}$, a function in $(\hat{x}, \hat{x}')$, holds. We use the nested boolean quantifier $\exists \hat{x}'$ in order to say 'there is *some* successor state.' Observe also the desired effect of $[\hat{x} := \hat{x}']$ performed on f^{ϕ}, thereby 'forcing' ϕ to be true at some next state[4].

The clause for EF is more complicated and involves the μ operator. Recall the equivalence

$$\mathrm{EF}\,\phi \equiv \phi \vee \mathrm{EX}\,\mathrm{EF}\,\phi. \tag{6.16}$$

[4] Exercise 6 on page 409 should give you a feel for how the semantics of $f[\hat{x} := \hat{x}']$ does not interfere with potential $\exists \hat{x}'$ or $\forall \hat{x}'$ quantifiers within f. For example, to evaluate $\rho \vDash (\exists \hat{x}'.f)[\hat{x} := \hat{x}']$, we evaluate $\rho[\hat{x} := \hat{x}'] \vDash \exists \hat{x}'.f$, which is true if we can find some values $(v_1, v_2, \dots, v_n) \in \{0,1\}^n$ such that $\rho[\hat{x} := \hat{x}'][x_1' \mapsto v_1][x_2' \mapsto v_2] \dots [x_n' \mapsto v_n] \vDash f$ is true. Observe that the resulting environment binds all x_i' to v_i, but for all other values it binds them according to $\rho[\hat{x} := \hat{x}']$; since the latter binds x_i to $\rho(x_i')$ which is the 'old' value of x_i', this is exactly what we desire in order to prevent a clash of variable names with the intended semantics.

Recall that an OBDD implementation synthesises formulas in a bottom-up fashion, so a reduced OBDD for $\exists \hat{x}'.f$ will not contain any x_i' nodes as its function does not depend on those variables. Thus, OBDDs also avoid such name clash problems.

Therefore, $f^{\mathrm{EF}\,\phi}$ has to be equivalent to $f^{\phi} + f^{\mathrm{EX}\,\mathrm{EF}\,\phi}$ which in turn is equivalent to $f^{\phi} + \exists \hat{x}'.\,(f^{\rightarrow} \cdot f^{\mathrm{EF}\,\phi}[\hat{x} := \hat{x}'])$. Now, since EF involves computing the *least* fixed point of the operator derived from the Equivalence (6.16), we obtain

$$f^{\mathrm{EF}\,\phi} \stackrel{\mathrm{def}}{=} \mu Z.\,(f^{\phi} + \exists \hat{x}'.\,(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}'])). \tag{6.17}$$

Note that the substitution $Z[\hat{x} := \hat{x}']$ means that the boolean function Z should be made to depend on the x_i' variables, rather than the x_i variables. This is because the evaluation of $\rho \vDash Z[\hat{x} := \hat{x}']$ results in $\rho[\hat{x} := \hat{x}'] \vDash Z$, where the latter valuation satisfies $\rho[\hat{x} := \hat{x}'](x_i) = \rho(x_i')$. Then, we use the modified valuation $\rho[\hat{x} := \hat{x}']$ to evaluate Z.

Since EF ϕ is equivalent to E[$\top$ U ϕ], we can generalise our coding of EF ϕ accordingly:

$$f^{\mathrm{E}[\phi \mathrm{U} \psi]} \stackrel{\mathrm{def}}{=} \mu Z.\,(f^{\psi} + f^{\phi} \cdot \exists \hat{x}'.\,(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}'])). \tag{6.18}$$

The coding of AF is similar to the one for EF in (6.17), except that 'for some' (boolean quantification $\exists \hat{x}'$) gets replaced by 'for all' (boolean quantification $\forall \hat{x}'$) and the 'conjunction' $f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}']$ turns into the 'implication' $\overline{f^{\rightarrow}} + Z[\hat{x} := \hat{x}']$:

$$f^{\mathrm{AF}\,\phi} \stackrel{\mathrm{def}}{=} \mu Z.\,(f^{\phi} + \forall \hat{x}'.\,(\overline{f^{\rightarrow}} + Z[\hat{x} := \hat{x}'])). \tag{6.19}$$

Notice how the semantics of $\mu Z.f$ in (6.12) reflects the intended meaning of the AF connective. The mth approximant of $f^{\mathrm{AF}\,\phi}$, which we write as $f_m^{\mathrm{AF}\,\phi}$, represents those states where all paths reach a ϕ-state within m steps.

This leaves us with coding EG, for then we have provided such a coding for an adequate fragment of CTL (recall Theorem 3.17 on page 216). Because EG involves computing greatest fixed points, we make use of the ν operator:

$$f^{\mathrm{EG}\,\phi} \stackrel{\mathrm{def}}{=} \nu Z.\,(f^{\phi} \cdot \exists \hat{x}'.\,(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}'])). \tag{6.20}$$

Observe that this does follow the logical structure of the semantics of EG: we need to show ϕ in the present state and then we have to find some successor state satisfying EG ϕ. The crucial point is that this obligation never ceases; this is exactly what we ensured in (6.14).

Let us see these codings in action on the model of Figure 6.24. We want to perform a symbolic model check of the formula EX $(x_1 \vee \neg x_2)$. You should verify, using e.g. the labelling algorithm from Chapter 3, that $[\![\mathrm{EX}\,(x_1 \vee \neg x_2)]\!] = \{s_1, s_2\}$. Our claim is that this set is computed symbolically by the resulting formula $f^{\mathrm{EX}\,(x_1 \vee \neg x_2)}$. First, we compute the formula

$f^{\rightarrow}$ which represents the transition relation →:

$$f^{\rightarrow} = (x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1)$$

where u is an input variable used to model the non-determinism (compare the form (6.6) for the transition relation in Section 6.3.4). Thus, we obtain

$$\begin{aligned} f^{\mathrm{EX}\,(x_1 \vee \neg x_2)} &= \exists x_1'.\exists x_2'.(f^{\rightarrow} \cdot f^{x_1 \vee \neg x_2}[\hat{x} := \hat{x}']) \\ &= \exists x_1'.\exists x_2'.((x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1) \cdot (x_1' + \overline{x}_2')). \end{aligned}$$

To see whether s_0 satisfies $\mathrm{EX}\,(x_1 \vee \neg x_2)$, we evaluate $\rho_0 \vDash f^{\mathrm{EX}\,(x_1 \vee \neg x_2)}$, where $\rho_0(x_1) = 1$ and $\rho_0(x_2) = 0$ (the value of $\rho_0(u)$ does not matter). We find that this does not hold, whence $s_0 \nvDash \mathrm{EX}\,(x_1 \vee \neg x_2)$. Likewise, we verify $s_1 \vDash \mathrm{EX}\,(x_1 \vee \neg x_2)$ by showing $\rho_1 \vDash f^{\mathrm{EX}\,(x_1 \vee \neg x_2)}$; and $s_2 \vDash \mathrm{EX}\,(x_1 \vee \neg x_2)$ by showing $\rho_2 \vDash f^{\mathrm{EX}\,(x_1 \vee \neg x_2)}$, where ρ_i is the valuation representing state s_i.

As a second example, we compute $f^{\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)}$ for the model in Figure 6.24. First, note that all three[5] states satisfy $\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)$, if we apply the labelling algorithm to the explicit model. Let us verify that the symbolic encoding matches this result. By (6.19), we have that $f^{\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)}$ equals

$$\mu Z. \left((\overline{x}_1 \cdot \overline{x}_2) + \forall x_1'.\forall x_2'.(x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1) \cdot Z[\hat{x} := \hat{x}']\right). \quad (6.21)$$

By (6.12), we have $\rho \vDash f^{\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)}$ iff $\rho \vDash f_m^{\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)}$ for some $m \geq 0$. Clearly, we have $\rho \nvDash f_0^{\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)}$. Now, $f_1^{\mathrm{AF}\,(\neg x_1 \wedge \neg x_2)}$ equals

$$((\overline{x}_1 \cdot \overline{x}_2) + \forall x_1'.\forall x_2'.(x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1) \cdot Z[\hat{x} := \hat{x}'])[0/Z].$$

Since $[0/Z]$ is a meta-operation, the latter formula is just

$$(\overline{x}_1 \cdot \overline{x}_2) + \forall x_1'.\forall x_2'.(x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1) \cdot 0[\hat{x} := \hat{x}'].$$

Thus, we need to evaluate the disjunction $(\overline{x}_1 \cdot \overline{x}_2) + \forall x_1'.\forall x_2'.(x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1) \cdot 0[\hat{x} := \hat{x}']$ at ρ. In particular, if $\rho(x_1) = 0$ and $\rho(x_2) = 0$, then $\rho \vDash \overline{x}_1 \cdot \overline{x}_2$ and so $\rho \vDash (\overline{x}_1 \cdot \overline{x}_2) + \forall x_1'.\forall x_2'.(x_1' \leftrightarrow \overline{x}_1 \cdot \overline{x}_2 \cdot u) \cdot (x_2' \leftrightarrow x_1) \cdot 0[\hat{x} := \hat{x}']$. Thus, $s_2 \vDash \mathrm{AF}\,(\neg x_1 \wedge \neg x_2)$ holds.

Similar reasoning establishes that the formula in (6.21) renders a correct coding for the remaining two states as well, which you are invited to verify as an exercise.

Symbolic model checking with fairness In Chapter 3, we sketched how SMV could use fairness assumptions which were not expressible entirely

[5] Since we have added the variable u, there are actually six states; they all satisfy the formula.

within CTL and its semantics. The addition of fairness could be achieved by restricting the ordinary CTL semantics to fair computation paths, or fair states. Formally, we were given a set $C = \{\psi_1, \psi_2, \dots, \psi_k\}$ of CTL formulas, called the *fairness constraints*, and we wanted to check whether $s \vDash \phi$ holds for a CTL formula ϕ and all initial states s, with the additional fairness constraints in C. Since $\bot$, $\neg$, $\wedge$, EX, EU and EG form an adequate set of connectives for CTL, we may restrict this discussion to only these operators. Clearly, the propositional connectives won't change their meaning with the addition of fairness constraints. Therefore, it suffices to provide symbolic codings for the fair connectives $\mathrm{E}_C\mathrm{X}$, $\mathrm{E}_C\mathrm{U}$ and $\mathrm{E}_C\mathrm{G}$ from Chapter 3. The key is to represent the set of fair states symbolically as a boolean formula fair defined as

$$\mathsf{fair} \stackrel{\mathrm{def}}{=} f^{\mathrm{E}_C\mathrm{G}\top} \tag{6.22}$$

which uses the (yet to be defined) function $f^{\mathrm{E}_C\mathrm{G}\,\phi}$ with $\top$ as an instance. Assuming that the coding of $f^{\mathrm{E}_C\mathrm{G}\,\phi}$ is correct, we see that fair computes 1 in a state s if, and only if, there is a fair path with respect to C that begins in s. We say that such an s is a *fair state*.

As for $\mathrm{E}_C\mathrm{X}$, note that $s \vDash \mathrm{E}_C\mathrm{X}\phi$ if, and only if, there is some next state s' with $s \to s'$ and $s' \vDash \phi$ such that s' is a fair state. This immediately renders

$$f^{\mathrm{E}_C\mathrm{X}\phi} \stackrel{\mathrm{def}}{=} \exists \hat{x}'.(f^{\to} \cdot (f^{\phi} \cdot \mathsf{fair})[\hat{x} := \hat{x}']). \tag{6.23}$$

Similarly, we obtain

$$f^{\mathrm{E}_C[\phi_1\mathrm{U}\phi_2]} \stackrel{\mathrm{def}}{=} \mu Z.\,(f^{\phi_2} \cdot \mathsf{fair} + f^{\phi_1} \cdot \exists \hat{x}'.\,(f^{\to} \cdot Z[\hat{x} := \hat{x}'])). \tag{6.24}$$

This leaves us with the task of coding $f^{\mathrm{E}_C\mathrm{G}\,\phi}$. It is this last connective which reveals the complexity of fairness checks at work. Because the coding of $f^{\mathrm{E}_C\mathrm{G}\,\phi}$ is rather complex, we proceed in steps. It is convenient to have the EX and EU functionality also at the level of boolean formulas directly. For example, if f is a boolean function in $\hat{x}$, then `checkEX` (f) codes the boolean formula which computes 1 for those vectors $\hat{x}$ which have a next state $\hat{x}'$ for which f computes 1:

$$\texttt{checkEX}\,(f) \stackrel{\mathrm{def}}{=} \exists \hat{x}'.(f^{\to} \cdot f[\hat{x} := \hat{x}']). \tag{6.25}$$

Thus, $f^{\mathrm{E}_C\mathrm{X}\phi}$ equals `checkEX` $(f^{\phi} \cdot \mathsf{fair})$. We proceed in the same way for functions f and g in n arguments $\hat{x}$ to obtain `checkEU` (f, g) which computes

1 at $\hat{x}$ if there is a path that realises the f U g pattern:

$$\texttt{checkEU}\,(f, g) \overset{\text{def}}{=} \mu Y.g + (f \cdot \texttt{checkEX}(Y)). \tag{6.26}$$

With this in place, we can code $f^{\mathrm{E}_C\mathrm{G}\phi}$ quite easily:

$$f^{\mathrm{E}_C\mathrm{G}\phi} \overset{\text{def}}{=} \nu Z.f^{\phi} \cdot \prod_{i=1}^{k} \texttt{checkEX}\,(\texttt{checkEU}\,(f^{\phi}, Z \cdot f^{\psi_i}) \cdot \mathsf{fair}). \tag{6.27}$$

Note that this coding has a least fixed point (checkEU) in the body of a greatest fixed point. This is computationally rather involved since the call of checkEU contains Z, the recursion variable of the outer greatest fixed point, as a free variable; thus these recursions are nested and inter-dependent; the recursions 'alternate.' Observe how this coding operates: to have a fair path from $\hat{x}$ on which ϕ holds globally, we need ϕ to hold at $\hat{x}$; and for all fairness constraints ψ_i there has to be a next state $\hat{x}'$, where the whole property is true again (enforced by the free Z) and each fairness constraint is realised eventually on that path. The recursion in Z constantly reiterates this reasoning, so if this function computes 1, then there is a path on which ϕ holds globally and where each ψ_i is true infinitely often.

6.5 Exercises

Exercises 6.1

1. Write down the truth tables for the boolean formulas in Example 6.2 on page 359. In your table, you may use 0 and 1, or F and T, whatever you prefer. What truth value does the boolean formula of item (4) on page 359 compute?
2. $\oplus$ is the exclusive-or: $x \oplus y \overset{\text{def}}{=} 1$ if the values of x and y are different; otherwise, $x \oplus y \overset{\text{def}}{=} 0$. Express this in propositional logic, i.e. find a formula ϕ having the same truth table as $\oplus$.

* 3. Write down a boolean formula $f(x, y)$ in terms of $\cdot$, $+$, $\bar{}$, 0 and 1, such that f has the same truth table as $p \to q$.

4. Write down a BNF for the syntax of boolean formulas based on the operations in Definition 6.1.

——

Exercises 6.2

* 1. Suppose we swap all dashed and solid lines in the binary decision tree of Figure 6.2. Write out the truth table of the resulting binary decision tree and find a formula for it.

* 2. Consider the following truth table:

p	q	r	ϕ
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

Write down a binary decision tree which represents the boolean function specified in this truth table.

3. Construct a binary decision tree for the boolean function specified in Figure 6.2, but now the root should be a y-node and its two successors should be x-nodes.
4. Consider the following boolean function given by its truth table:

x	y	z	$f(x, y, z)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	1

(a) Construct a binary decision tree for $f(x, y, z)$ such that the root is an x-node followed by y- and then z-nodes.

(b) Construct another binary decision tree for $f(x, y, z)$, but now let its root be a z-node followed by y- and then x-nodes.

5. Let T be a binary decision tree for a boolean function $f(x_1, x_2, \ldots, x_n)$ of n boolean variables. Suppose that every variable occurs exactly once as one travels down on any path of the tree T. Use mathematical induction to show that T has $2^{n+1} - 1$ nodes.

Exercises 6.3

* 1. Explain why all reductions C1–C3 (page 363) on a BDD B result in BDDs which still represent the same function as B.

2. Consider the BDD in Figure 6.7.

* (a) Specify the truth table for the boolean function $f(x, y, z)$ represented by this BDD.

(b) Find a BDD for that function which does not have multiple occurrences of variables along any path.

3. Let f be the function represented by the BDD of Figure 6.3(b). Using also the BDDs B_0, B_1 and B_x illustrated in Figure 6.6, find BDDs representing
   (a) $f \cdot x$
   (b) $x + f$
   (c) $\overline{f \cdot 0}$
   (d) $f \cdot 1$.

Exercises 6.4

1. Figure 6.9 (page 367) shows a BDD with ordering $[x, y, z]$.
   * (a) Find an equivalent reduced BDD with ordering $[z, y, x]$. (Hint: find first the decision tree with the ordering $[z, y, x]$, and then reduce it using C1–C3.)
   (b) Carry out the same construction process for the variable ordering $[y, z, x]$. Does the reduced BDD have more or fewer nodes than the ones for the orderings $[x, y, z]$ and $[z, y, x]$?
2. Consider the BDDs in Figures 6.4–6.10. Determine which of them are OBDDs. If you find an OBDD, you need to specify a list of its boolean variables without double occurrences which demonstrates that ordering.
3. Consider the following boolean formulas. Compute their unique reduced OBDDs with respect to the ordering $[x, y, z]$. It is advisable to first compute a binary decision tree and then to perform the removal of redundancies.
   (a) $f(x, y) \stackrel{\text{def}}{=} x \cdot y$
   * (b) $f(x, y) \stackrel{\text{def}}{=} x + y$
   (c) $f(x, y) \stackrel{\text{def}}{=} x \oplus y$
   * (d) $f(x, y, z) \stackrel{\text{def}}{=} (x \oplus y) \cdot (\overline{x} + z)$.
4. Recall the derived connective $\phi \leftrightarrow \psi$ from Chapter 1 saying that for all valuations ϕ is true if, and only if, ψ is true.
   (a) Define this operator for boolean formulas using the basic operations $\cdot$, $+$, $\oplus$ and $\bar{\ }$ from Definition 6.1.
   (b) Draw a reduced OBDD for the formula $g(x, y) \stackrel{\text{def}}{=} x \leftrightarrow y$ using the ordering $[y, x]$.
5. Consider the even parity function introduced at the end of the last section.
   (a) Define the odd parity function $f_{\text{odd}}(x_1, x_2, \dots, x_n)$.
   (b) Draw an OBDD for the odd parity function for $n = 5$ and the ordering $[x_3, x_5, x_1, x_4, x_2]$. Would the overall structure of this OBDD change if you changed the ordering?
   (c) Show that $f_{\text{even}}(x_1, x_2, \dots, x_n)$ and $\overline{f_{\text{odd}}(x_1, x_2, \dots, x_n)}$ denote the same boolean function.
6. Use Theorem 6.7 (page 368) to show that, if the reductions C1–C3 are applied until no more reduction is possible, the result is independent of the order in which they were applied.

Exercises 6.5

1. Given the boolean formula $f(x_1, x_2, x_3) \stackrel{\text{def}}{=} x_1 \cdot (x_2 + \overline{x_3})$, compute its reduced OBDD for the following orderings:
   (a) $[x_1, x_2, x_3]$
   (b) $[x_3, x_1, x_2]$
   (c) $[x_3, x_2, x_1]$.
2. Compute the reduced OBDD for $f(x, y, z) = x \cdot (z + \overline{z}) + \overline{y} \cdot \overline{x}$ in any ordering you like. Is there a z-node in that reduced OBDD?
3. Consider the boolean formula $f(x, y, z) \stackrel{\text{def}}{=} (\overline{x} + y + \overline{z}) \cdot (x + \overline{y} + z) \cdot (x + y)$. For the variable orderings below, compute the (unique) reduced OBDD B_f of f with respect to that ordering. It is best to write down the binary decision tree for that ordering and then to apply all possible reductions.
   (a) $[x, y, z]$.
   (b) $[y, x, z]$.
   (c) $[z, x, y]$.
   (d) Find an ordering of variables for which the resulting reduced OBDD B_f has a minimal number of edges; i.e. there is no ordering for which the corresponding B_f has fewer edges. (How many possible orderings for x, y and z are there?)
4. Given the truth table

x	y	z	$f(x, y, z)$
1	1	1	0
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	0
0	1	0	1
0	0	1	0
0	0	0	1

   compute the reduced OBDD with respect to the following ordering of variables:
   (a) $[x, y, z]$
   (b) $[z, y, x]$
   (c) $[y, z, x]$
   (d) $[x, z, y]$.
5. Given the ordering $[p, q, r]$, compute the reduced BDDs for $p \wedge (q \vee r)$ and $(p \wedge q) \vee (p \wedge r)$ and explain why they are identical.
6. * Consider the BDD in Figure 6.11 (page 370).
   (a) Construct its truth table.
   (b) Compute its conjunctive normal form.
   (c) Compare the length of that normal form with the size of the BDD. What is your assessment?

Exercises 6.6

1. Perform the execution of `reduce` on the following OBDDs:
   (a) The binary decision tree for
      i. $x \oplus y$
      ii. $x \cdot y$
      iii. $x + y$
      iv. $x \leftrightarrow y$.
   (b) The OBDD in Figure 6.2 (page 361).
 * (c) The OBDD in Figure 6.4 (page 363).

——

Exercises 6.7

1. Recall the Shannon expansion in (6.1) on page 374. Suppose that x does not occur in f at all. Why does (6.1) still hold?
2. Let $f(x, y, z) \overset{\text{def}}{=} y + \overline{z} \cdot x + z \cdot \overline{y} + y \cdot x$ be a boolean formula. Compute f's Shannon expansion with respect to
   (a) x
   (b) y
   (c) z.
3. Show that boolean formulas f and g are semantically equivalent if, and only if, the boolean formula $(\overline{f} + g) \cdot (f + \overline{g})$ computes 1 for all possible assignments of 0s and 1s to their variables.
4. We may use the Shannon expansion to define formally how BDDs determine boolean functions. Let B be a BDD. It is intuitively clear that B determines a unique boolean function. Formally, we compute a function f_n inductively (bottom-up) for all nodes n of B:
   – If n is a terminal node labelled 0, then f_n is the constant 0 function.
   – Dually, if n is a terminal 1-node, then f_n is the constant 1 function.
   – If n is a non-terminal node labelled x, then we already have defined the boolean functions $f_{\mathrm{lo}(n)}$ and $f_{\mathrm{hi}(n)}$ and set f_n to be $\overline{x} \cdot f_{\mathrm{lo}(n)} + x \cdot f_{\mathrm{hi}(n)}$.

   If i is the initial node of B, then f_i is the boolean function represented by B. Observe that we could apply this definition as a symbolic evaluation of B resulting in a boolean formula. For example, the BDD of Figure 6.3(b) renders $\overline{x} \cdot (\overline{y} \cdot 1 + y \cdot 0) + x \cdot 0$. Compute the boolean formulas obtained in this way for the following BDDs:
   (a) the BDD in Figure 6.5(b) (page 364)
   (b) the BDDs in Figure 6.6 (page 365)
   (c) the BDD in Figure 6.11 (page 370).
* 5. Consider a ternary (= takes three arguments) boolean connective $f \rightarrow (g, h)$ which is equivalent to g when f is true; otherwise, it is equivalent to h.
   (a) Define this connective using any of the operators $+$, $\cdot$, $\oplus$ or $\overline{\ }$.
   (b) Recall exercise 4. Use the ternary operator above to write f_n as an expression of $f_{\mathrm{lo}(n)}$, $f_{\mathrm{hi}(n)}$ and its label x.

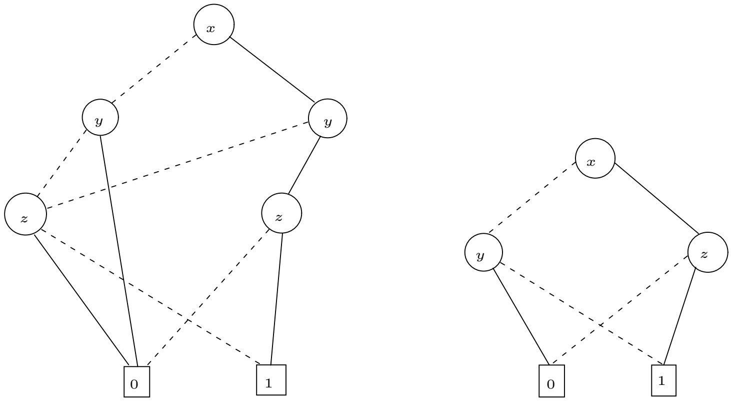


Figure 6.30. The reduced OBDDs B_f and B_g (see exercises).

(c) Use mathematical induction (on what?) to prove that, if the root of f_n is an x-node, then f_n is independent of any y which comes before x in an assumed variable ordering.

6. Explain why `apply` (op, B_f, B_g), where B_f and B_g have compatible ordering, produces an OBDD with an ordering compatible with that of B_f and B_g.
7. Explain why the four cases of the control structure for `apply` are exhaustive, i.e. there are no other possible cases in its execution.
8. Consider the reduced OBDDs B_f and B_g in Figure 6.30. Recall that, in order to compute the reduced OBDD for f op g, you need to
   – construct the tree showing the recursive descent of `apply` (op, B_f, B_g) as done in Figure 6.16;
   – use that tree to simulate `apply` (op, B_f, B_g); and
   – reduce, if necessary, the resulting OBDD.

   Perform these steps on the OBDDs of Figure 6.30 for the operation 'op' being
   (a) $+$
   (b) $\oplus$
   (c) $\cdot$
9. Let B_f be the OBDD in Figure 6.11 (page 370). Compute `apply` ($\oplus, B_f, B_1$) and reduce the resulting OBDD. If you did everything correctly, then this OBDD should be isomorphic to the one obtained from swapping 0- and 1-nodes in Figure 6.11.

* 10. Consider the OBDD B_c in Figure 6.31 which represents the 'don't care' conditions for comparing the boolean functions f and g represented in Figure 6.30. This means that we want to compare whether f and g are equal for all values of variables except those for which c is true (i.e. we 'don't care' when c is true).
   (a) Show that the boolean formula $(\overline{f} \oplus g) + c$ is valid (always computes 1) if, and only if, f and g are equivalent on all values for which c evaluates to 0.

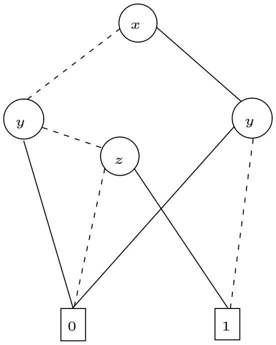


Figure 6.31. The reduced OBDD B_c representing the 'don't care' conditions for the equivalence test of the OBDDs in Figure 6.30.

(b) Proceed in three steps as in exercise 8 on page 403 to compute the reduced OBDD for $(\overline{f} \oplus g) + c$ from the OBDDs for f, g and c. Which call to `apply` needs to be first?

11. We say that $v \in \{0, 1\}$ is a (left)-controlling value for the operation op, if either $v \operatorname{op} x = 1$ or $v \operatorname{op} x = 0$ for all values of x. We say that v is a controlling value if it is a left- and right-controlling value.
    (a) Define the notion of a right-controlling value.
    (b) Give examples of operations with controlling values.
    (c) Describe informally how `apply` can be optimised when op has a controlling value.
    (d) Could one still do some optimisation if op had only a left- or right-controlling value?
12. We showed that the worst-time complexity of `apply` is $O(|B_f| \cdot |B_g|)$. Show that this upper bound is hard, i.e. it cannot be improved:
    (a) Consider the functions $f(x_1, x_2, \dots, x_{2n+2m}) \overset{\text{def}}{=} x_1 \cdot x_{n+m+1} + \cdots + x_n \cdot x_{2n+m}$ and $g(x_1, x_2, \dots, x_{2n+2m}) \overset{\text{def}}{=} x_{n+1} \cdot x_{2n+m+1} + \cdots + x_{n+m} \cdot x_{2n+2m}$ which are in sum-of-product form. Compute the sum-of-product form of $f + g$.
    (b) Choose the ordering $[x_1, x_2, \dots, x_{2n+2m}]$ and argue that the OBDDs B_f and B_g have 2^{n+1} and 2^{m+1} edges, respectively.
    (c) Use the result from part (a) to conclude that B_{f+g} has 2^{n+m+1} edges, i.e. $0.5 \cdot |B_f| \cdot |B_g|$.

——

Exercises 6.8

1. Let f be the reduced OBDD represented in Figure 6.5(b) (page 364). Compute the reduced OBDD for the restrictions:
   (a) $f[0/x]$
   * (b) $f[1/x]$

(c) $f[1/y]$

* (d) $f[0/z]$.

* 2. Suppose that we intend to modify the algorithm `restrict` so that it is capable of computing reduced OBDDs for a general composition $f[g/x]$.
   (a) Generalise Equation (6.1) to reflect the intuitive meaning of the operation $[g/x]$.
   (b) What fact about OBDDs causes problems for computing this composition directly?
   (c) How can we compute this composition given the algorithms discussed so far?

3. We define read-1-BDDs as BDDs B where each boolean variable occurs at most once on any evaluation path of B. In particular, read-1-BDDs need not possess an ordering on their boolean variables. Clearly, every OBDD is a read-1-BDD; but not every read-1-BDD is an OBDD (see Figure 6.10). In Figure 6.18 we see a BDD which is not a read-1-BDD; the path for $(x, y, z) \Rightarrow (1, 0, 1)$ 'reads' the value of x twice.
   Critically assess the implementation of boolean formulas via OBDDs to see which implementation details could be carried out for read-1-BDDs as well. Which implementation aspects would be problematic?

4. (For those who have had a course on finite automata.) Every boolean function f in n arguments can be viewed as a subset L_f of $\{0, 1\}^n$; defined to be the set of all those bit vectors $(v_1, v_2, \dots, v_n)$ for which f computes 1. Since this is a finite set, L_f is a regular language and has therefore a deterministic finite automaton with a minimal number of states which accepts L_f. Can you match some of the OBDD operations with those known for finite automata? How close is the correspondence? (You may have to consider non-reduced OBDDs.)

5. (a) Show that every boolean function in n arguments can be represented as a boolean formula of the grammar

$$f ::= 0 \mid x \mid \overline{f} \mid f_1 + f_2.$$

   (b) Why does this also imply that every such function can be represented by a reduced OBDD in any variable ordering?

6. Use mathematical induction on n to prove that there are exactly $2^{(2^n)}$ many different boolean functions in n arguments.

Exercises 6.9

1. Use the `exists` algorithm to compute the OBDDs for
   (a) $\exists x_3.f$, given the OBDD for f in Figure 6.11 (page 370)
   (b) $\forall y.g$, given the OBDD for g in Figure 6.9 (page 367)
   (c) $\exists x_2.\exists x_3.x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$.

2. Let f be a boolean function depending on n variables.
   (a) Show:

i. The formula $\exists x.f$ depends on all those variables that f depends upon, except x.

ii. If f computes to 1 with respect to a valuation ρ, then $\exists x.f$ computes 1 with respect to the same valuation.

iii. If $\exists x.f$ computes to 1 with respect to a valuation ρ, then there is a valuation ρ' for f which agrees with ρ for all variables other than x such that f computes to 1 under ρ'.

(b) Can the statements above be shown for the function value 0?

3. Let ϕ be a boolean formula.

* (a) Show that ϕ is satisfiable if, and only if, $\exists x.\phi$ is satisfiable.

(b) Show that ϕ is valid if, and only if, $\forall x.\phi$ is valid.

(c) Generalise the two facts above to nested quantifications $\exists \hat{x}$ and $\forall \hat{x}$. (Use induction on the number of quantified variables.)

4. Show that $\forall \hat{x}.f$ and $\overline{\exists \hat{x}.\overline{f}}$ are semantically equivalent. Use induction on the number of arguments in the vector $\hat{x}$.

Exercises 6.10
(For those who know about complexity classes.)

1. Show that 3SAT can be reduced to nested existential boolean quantification. Given an instance of 3SAT, we may think of it as a boolean formula f in product-of-sums form $g_1 \cdot g_2 \cdot \dots \cdot g_n$, where each g_i is of the form $(l_1 + l_2 + l_3)$ with each l_j being a boolean variable or its complementation. For example, f could be $(x + \overline{y} + z) \cdot (x_5 + x + \overline{x}_7) \cdot (\overline{x}_2 + z + x) \cdot (x_4 + \overline{x}_2 + \overline{x}_4)$.

(a) Show that you can represent each function g_i with an OBDD of no more than three non-terminals, independently of the chosen ordering.

(b) Introduce n new boolean variables z_1, z_2, ..., z_n. We write $\sum_{1\leq i\leq n} f_i$ for the expression $f_1 + f_2 + \cdots + f_n$ and $\prod_{1\leq i\leq n} f_i$ for $f_1 \cdot f_2 \cdot \dots \cdot f_n$. Consider the boolean formula h, defined as

$$\sum_{1\leq i\leq n} \left(\overline{g}_i \cdot z_i \cdot \prod_{1\leq j<i} \overline{z}_j \right). \tag{6.28}$$

Choose any ordering of variables whose list begins as in $[z_1, z_2, \dots, z_n, \dots]$. Draw the OBDD for h (draw only the root nodes for $\overline{g}_i$).

(c) Argue that the OBDD above has at most $4n$ non-terminal nodes.

(d) Show that f is satisfiable if, and only if, the OBDD for $\exists z_1.\exists z_2. \dots .\exists z_n.h$ is not equal to B_1.

(e) Explain why the last item shows a reduction of 3SAT to nested existential quantification.

2. Show that the problem of finding an optimal ordering for representing boolean functions as OBDDs is in coNP.

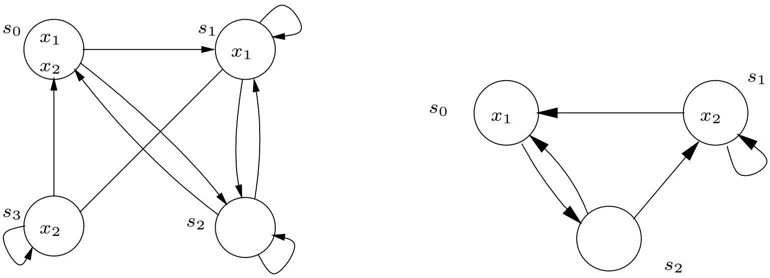


Figure 6.32. (a) A CTL model with four states. (b) A CTL model with three states.

3. Recall that $\exists x.f$ is defined as $f[1/x] + f[0/x]$. Since we have efficient algorithms for restriction and $+$, we obtain hereby an efficient algorithm for $\exists z_1 \ldots . \exists z_n.f$. Thus, P equals NP! What is wrong with this argument?

Exercises 6.11

* 1. Consider the CTL model in Figure 6.32(a). Using the ordering $[x_1, x_2]$, draw the OBDD for the subsets $\{s_0, s_1\}$ and $\{s_0, s_2\}$.
2. Consider the CTL model in Figure 6.32(b). Because the number of states is not an exact power of 2, there are more than one OBDDs representing any given set of states. Using again the ordering $[x_1, x_2]$, draw all possible OBDDs for the subsets $\{s_0, s_1\}$ and $\{s_0, s_2\}$.

Exercises 6.12

1. Consider the CTL model in Figure 6.32(a).
   (a) Work out the truth table for the transition relation, ordering the columns $[x_1, x_1', x_2, x_2']$. There should be as many 1s in the final column as there are arrows in the transition relation. There is no freedom in the representation in this case, since the number of states is an exact power of 2.
   (b) Draw the OBDD for this transition relation, using the variable ordering $[x_1, x_1', x_2, x_2']$.
2. Apply the algorithm of Section 3.6.1, but now interpreted over OBDDs in the ordering $[x_1, x_2]$, to compute the set of states of the CTL model in Figure 6.32(b) which satisfy
   (a) $\mathrm{AG}\,(x_1 \lor \neg x_2)$
   (b) $\mathrm{E}[x_2 \mathrm{~U~} x_1]$.

   Show the OBDDs which are computed along the way.
3. Explain why `exists`$(\hat{x}', $ `apply`$(\cdot, B_{\to}, B_{X'}))$ faithfully implements the meaning of $\mathrm{pre}_{\exists}(X)$.

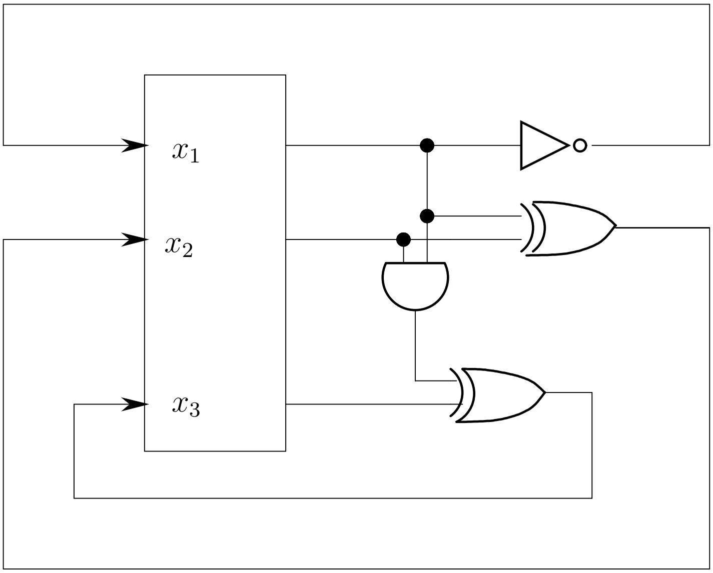


Figure 6.33. A synchronous circuit for a modulo 8 counter.

Exercises 6.13

1. (a) Simulate the evolution of the circuit in Figure 6.29 (page 389) with initial state 01. What do you think that it computes?
   (b) Write down the explicit CTL model $(S, \to, L)$ for this circuit.
2. Consider the sequential synchronous circuit in Figure 6.33.
   (a) Construct the functions f_i for $i = 1, 2, 3$.
   (b) Code the function $f^{\to}$.
   (c) Recall from Chapter 2 that $(\exists x.\phi) \wedge \psi$ is semantically equivalent to $\exists x.(\phi \wedge \psi)$ if x is not free in ψ.
      i. Why is this also true in our setting of boolean formulas?
      ii. Apply this law to push the $\exists$ quantifications in $f^{\to}$ as far inwards as possible. This is an often useful optimisation in checking synchronous circuits.
3. Consider the boolean formula for the 2-bit comparator:

$$f(x_1, x_2, y_1, y_2) \stackrel{\text{def}}{=} (x_1 \leftrightarrow y_1) \cdot (x_2 \leftrightarrow y_2).$$

   (a) Draw its OBDD for the ordering $[x_1, y_1, x_2, y_2]$.
   (b) Draw its OBDD for the ordering $[x_1, x_2, y_1, y_2]$ and compare that with the one above.
4. (a) Can you use (6.6) from page 388 to code the transition relation $\to$ of the model in Figure 6.24 on page 384?
   (b) Can you do it with equation (6.9) from page 390?
   (c) With equation (6.8) from page 389?

Exercises 6.14

1. Let ρ be the valuation for which $(x, y, z) \Rightarrow (0, 1, 1)$. Compute whether $\rho \vDash f$ holds for the following boolean formulas:
   (a) $x \cdot (y + \overline{z} \cdot (y \oplus x))$
   (b) $\exists x.(y \cdot (x + z + \overline{y}) + x \cdot \overline{y})$
   (c) $\forall x.(y \cdot (x + z + \overline{y}) + x \cdot \overline{y})$
   (d) $\exists z.(x \cdot \overline{z} + \forall x.((y + (x + \overline{x}) \cdot z)))$
   * (e) $\forall x.(y + \overline{z})$.
* 2. Use (6.14) from page 393 and the definition of the satisfaction relation for formulas of the relational mu-calculus to prove $\rho \vDash \nu Z.Z$ for all valuations ρ. In this case, f equals Z and you need to show (6.14) by mathematical induction on $m \geq 0$.
3. An implementation which decides $\vDash$ and $\nvDash$ for the relational mu-calculus obviously cannot represent valuations which assign semantic values 0 or 1 to all, i.e. infinitely many variables. Thus, it makes sense to consider $\vDash$ as a relation between pairs (ρ, f), where ρ only assigns semantic values to all free variables of f.
   (a) Assume that νZ and μZ, $\exists x$, $\forall x$, and $[\hat{x} := \hat{x}']$ are binding constructs similar to the quantifiers in predicate logic. Define formally the set of free variables for a formula f of the relational mu-calculus. (Hint: You should define this by structural induction on f. Also, which variables get bound in $f[\hat{x} := \hat{x}']$?)
   (b) Recall the notion of t being free for x in ϕ which we discussed in Section 2.2.4 Define what 'g is free for Z in f' should mean and find an example, where g is not free for Z in f.
   (c) Explain informally why we can decide whether $\rho \vDash f$ holds, provided that ρ assigns values 0 or 1 to all free variables of f. Explain why this answer will be independent of what ρ does to variables which are bound in f. Why is this relevant for an implementation framework?
4. Let ρ be the valuation for which $(x, x', y, y') \Rightarrow (0, 1, 1, 1)$. Determine whether $\rho \vDash f$ holds for the following formulas f (recall that we write $f \leftrightarrow g$ as an abbreviation for $\overline{f} \oplus g$, meaning that f computes 1 iff g computes 1):
   (a) $\exists x.(x' \leftrightarrow (\overline{y} + y' \cdot x))$
   (b) $\forall x.(x' \leftrightarrow (\overline{y} + y' \cdot x))$
   (c) $\exists x'.(x' \leftrightarrow (\overline{y} + y' \cdot x))$
   (d) $\forall x'.(x' \leftrightarrow (\overline{y} + y' \cdot x))$.
5. Let ρ be a valuation with $\rho(x_1') = 1$ and $\rho(x_2') = 0$. Determine whether $\rho \vDash f$ holds for the following:
   (a) $\overline{x}_1[\hat{x} := \hat{x}']$
   (b) $(x_1 + \overline{x}_2)[\hat{x} := \hat{x}']$
   (c) $(\overline{x}_1 \cdot \overline{x}_2)[\hat{x} := \hat{x}']$.
6. Evaluate $\rho \vDash (\exists x_1.(x_1 + \overline{x}_2))[\hat{x} := \hat{x}']$ and explain how the valuation ρ changes in that process. In particular, $[\hat{x} := \hat{x}']$ replaces x_i by x_i', but why does this not interfere with the binding quantifier $\exists x_1$?

7. (a) How would you define the notion of semantic entailment for the relational mu-calculus?
   (b) Define formally when two formulas of the relational mu-calculus are semantically equivalent.

Exercises 6.15

1. Using the model of Figure 6.24 (page 384), determine whether $\rho \vDash f^{\mathrm{EX}\,(x_1 \vee \neg x_2)}$ holds, where ρ is
   (a) $(x_1, x_2) \Rightarrow (1, 0)$
   (b) $(x_1, x_2) \Rightarrow (0, 1)$
   (c) $(x_1, x_2) \Rightarrow (0, 0)$.
2. Let S be $\{s_0, s_1\}$, with $s_0 \to s_0$, $s_0 \to s_1$ and $s_1 \to s_0$ as possible transitions and $L(s_0) = \{x_1\}$ and $L(s_1) = \emptyset$. Compute the boolean function $f^{\mathrm{EX}\,(\mathrm{EX}\,\neg x_1)}$.
3. Equations (6.17) (page 395), (6.19) and (6.20) define $f^{\mathrm{EF}\,\phi}$, $f^{\mathrm{AF}\,\phi}$ and $f^{\mathrm{EG}\,\phi}$. Write down a similar equation to define $f^{\mathrm{AG}\,\phi}$.
4. Define a direct coding $f^{\mathrm{AU}\,\phi}$ by modifying (6.18) appropriately.
5. Mimic the example checks on page 396 for the connective AU: consider the model of Figure 6.24 (page 384). Since $[\![\mathrm{E}[(x_1 \vee x_2)\ \mathrm{U}\ (\neg x_1 \wedge \neg x_2)]]\!]$ equals the entire state set $\{s_0, s_1, s_2\}$, your coding of $f^{\mathrm{E}[x_1 \vee x_2 \mathrm{U} \neg x_1 \wedge \neg x_2]}$ is correct if it computes 1 for all bit vectors different from $(1, 1)$.
   (a) Verify that your coding is indeed correct.
   (b) Find a boolean formula without fixed points which is semantically equivalent to $f^{\mathrm{E}[(x_1 \vee x_2)\mathrm{U}(\neg x_1 \wedge \neg x_2)]}$.
6. (a) Use (6.20) on page 395 to compute $f^{\mathrm{EG}\,\neg x_1}$ for the model in Figure 6.24.
   (b) Show that $f^{\mathrm{EG}\,\neg x_1}$ faithfully models the set of all states which satisfy $\mathrm{EG}\,\neg x_1$.
7. In the grammar (6.10) for the relational mu-calculus on page 390, it was stated that, in the formulas $\mu Z.f$ and $\nu Z.f$, any occurrence of Z in f is required to fall within an even number of complementation symbols $\bar{\ }$. What happens if we drop this requirement?
   (a) Consider the expression $\mu Z.\overline{Z}$. We already saw that our relation ρ is total in the sense that either $\rho \vDash f$ or $\rho \nvDash f$ holds for all choices of valuations ρ and relational mu-calculus formulas f. But formulas like $\mu Z.\overline{Z}$ are not formally monotone. Let ρ be any valuation. Use mutual mathematical induction to show:
      i. $\rho \nvDash \mu_m Z.\overline{Z}$ for all even numbers $m \geq 0$
      ii. $\rho \vDash \mu_m Z.\overline{Z}$ for all odd numbers $m \geq 1$

      Infer from these two items that $\rho \vDash \mu Z.\overline{Z}$ holds according to (6.12).
   (b) Consider any environment ρ. Use mathematical induction on m (and maybe an analysis on ρ) to show:

      If $\rho \vDash \mu_m Z.(x_1 + \overline{x_2 \cdot \overline{Z}})$ for some $m \geq 0$, then $\rho \vDash \mu_k Z.(x_1 + \overline{x_2 \cdot \overline{Z}})$ for all $k \geq m$.

(c) In general, if f is formally monotone in Z then $\rho \vDash \mu_m Z.f$ implies $\rho \vDash \mu_{m+1} Z.f$. Can you state a similar property for the greatest fixed-point operator ν?

8. Given the CTL model for the circuit in Figure 6.29 (page 389):
 * (a) code the function $f^{\mathrm{EX}\,(x_1 \wedge \neg x_2)}$
 (b) code the function $f^{\mathrm{AG}\,(\mathrm{AF}\,\neg x_1 \wedge \neg x_2)}$
 * (c) find a boolean formula without any fixed points which is semantically equivalent to $f^{\mathrm{AG}\,(\mathrm{AF}\,\neg x_1 \wedge \neg x_2)}$.
9. Consider the sequential synchronous circuit in Figure 6.33 (page 408). Evaluate $\rho \vDash f^{\mathrm{EX}\,x_2}$, where ρ equals
 (a) $(x_1, x_2, x_3) \Rightarrow (1, 0, 1)$
 (b) $(x_1, x_2, x_3) \Rightarrow (0, 1, 0)$.
10. Prove

Theorem 6.19 Given a coding for a finite CTL model, let ϕ be a CTL formula from an adequate fragment. Then $[\![\phi]\!]$ corresponds to the set of valuations ρ such that $\rho \vDash f^{\phi}$.

by structural induction on ϕ. You may first want to show that the evaluation of $\rho \vDash f^{\phi}$ depends only on the values $\rho(x_i)$, i.e. it does not matter what ρ assigns to x_i' or Z.

11. Argue that Theorem 6.19 above remains valid for arbitrary CTL formulas as long as we translate formulas ϕ which are not in the adequate fragment into semantically equivalent formulas ψ in that fragment and define f^{ϕ} to be f^{ψ}.
12. Derive the formula $f^{\mathrm{AF}\,(\neg x_1 \wedge x_2)}$ for the model in Figure 6.32(b) on page 407 and evaluate it for the valuation corresponding to state s_2 to determine whether $s_2 \vDash \mathrm{AF}\,(\neg x_1 \wedge x_2)$ holds.
13. Repeat the last exercise with $f^{\mathrm{E}[x_1 \vee \neg x_2 \mathrm{U} x_1]}$.
14. Recall the way the two labelling algorithms operate in Chapter 3. Does our symbolic coding mimic either or both of them, or neither?

Exercises 6.16

1. Consider the equations in (6.22) and (6.27). The former defines fair in terms of $f^{\mathrm{E}_C\mathrm{G}\top}$, whereas the latter defines $f^{\mathrm{E}_C\mathrm{G}\,\phi}$ for general ϕ. Why is this unproblematic, i.e. non-circular?
2. Given a fixed CTL model $\mathcal{M} = (S, \rightarrow, L)$, we saw how to code formulas f^{ϕ} representing the set of states $s \in S$ with $s \vDash \phi$, ϕ being a CTL formula of an adequate fragment.
 (a) Assume the coding without consideration of simple fairness constraints. Use structural induction on the CTL formula ϕ to show that
  i. the free variables of f^{ϕ} are among $\hat{x}$, where the latter is the vector of boolean variables which code states $s \in S$;
  ii. all fixed-point subformulas of f^{ϕ} are formally monotone.

(b) Show these two assertions if f^{ϕ} also encodes simple fairness constraints.

3. Consider the pseudo-code for the function SAT on page 227. We now want to modify it so that the resulting output is not a set, or an OBDD, but a formula of the relational mu-calculus; thus, we complete the table in Figure 6.22 on page 380 to give formulas of the relational mu-calculus. For example, the output for $\top$ would be 1 and the output for $\mathrm{EU}\,\psi$ would be a recursive call to SAT informed by (6.18). Do you have a need for a separate function which handles least or greatest fixed points?
4. (a) Write pseudo-code for a function $\mathtt{SAT}_{\mathrm{rel\_mu}}$ which takes as input a formula of the relational mu-calculus, f, and synthesises an OBDD B_f, representing f. Assume that there are no fixed-point subexpressions of f such that their recursive body contains a recursion variable of an outwards fixed point. Thus, the formula in (6.27) is not allowed. The fixed-point operators μ and ν require separate subfunctions which iterate the fixed-point meaning informed by (6.12), respectively (6.14). Some of your clauses may need further comment. E.g. how do you handle the constructor $[\hat{x} := \hat{x}']$?
   (b) Explain what goes wrong if the input to your code is the formula in (6.27).
5. If f is a formula with a vector of n free boolean variables $\hat{x}$, then the iteration of $\mu Z.f$, whether as OBDD implementation, or as in (6.12), may require up to 2^n recursive unfoldings to compute its meaning. Clearly, this is unacceptable. Given the symbolic encoding of a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a set $I \subseteq S$ of initial states, we seek a formula that represents all states which are reachable from I on some finite computation path in $\mathcal{M}$. Using the extended Until operator in (6.26), we may express this as $\mathtt{checkEU}\,(f^I, \top)$, where f^I is the characteristic function of I. We can 'speed up' this iterative process with a technique called 'iterative squaring':

$$\mu Y.(f^{\rightarrow} + \exists \hat{w}.(Y[\hat{x}' := \hat{w}] \cdot Y[\hat{x} := \hat{w}])). \tag{6.29}$$

Note that this formula depends on the same boolean variables as $f^{\rightarrow}$, i.e. the pair $(\hat{x}, \hat{x}')$. Explain informally:

> If we apply (6.12) m times to the formula in (6.29), then this has the same semantic 'effect' as applying this rule 2^m times to $\mathtt{checkEU}\,(f^{\rightarrow}, \top)$.

Thus, one may first compute the set of states reachable from any initial state and then restrict model checking to those states. Note that this reduction does not alter the semantics of $s \vDash \phi$ for initial states s, so it is a sound technique; it sometimes improves, other times worsens, the performance of symbolic model checks.

6.6 Bibliographic notes

Ordered binary decision diagrams are due to R. E. Bryant [Bry86]. Binary decision diagrams were introduced by C. Y. Lee [Lee59] and S. B. Akers [Ake78]. For a nice survey of these ideas see [Bry92]. For the limitations of OBDDs as models for integer multiplication as well as interesting connections to VLSI design see [Bry91]. A general introduction to the topic of computational complexity and its tight connections to logic can be found in [Pap94]. The modal mu-calculus was invented by D. Kozen [Koz83]; for more on that logic and its application to specifications and verification see [Bra91].

The use of BDDs in model checking was proposed by the team of authors J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and J. Hwang [BCM$^+$90, CGL93, McM93].

Bibliography

Ake78. S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

AO91. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

Bac86. R. C. Backhouse. *Program Construction and Verification.* Prentice Hall, 1986.

BCCZ99. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, 1999.

BCM$^+$90. J. R. Burch, J. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1990.

BEKV94. K. Broda, S. Eisenbach, H. Khoshnevisan, and S. Vickers. *Reasoned Programming.* Prentice Hall, 1994.

BJ80. G. Boolos and R. Jeffrey. *Computability and Logic.* Cambridge University Press, 2nd edition, 1980.

Boo54. G. Boole. *An Investigation of the Laws of Thought.* Dover, New York, 1854.

Bra91. J. C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, Boston, 1991.

Bry86. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Compilers*, C-35(8), 1986.

Bry91. R. E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

Bry92. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

CE81. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs Workshop*, number 131 in LNCS. Springer Verlag, 1981.

CGL93. E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, number 803 in Lecture Notes in Computer Science, pages 124–175. Springer Verlag, 1993.

CGL94. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

CGP99. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

Che80. B. F. Chellas. *Modal Logic – an Introduction.* Cambridge University Press, 1980.

Dam96. D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Institute for Programming Research and Algorithmics. Eindhoven University of Technology, July 1996.

Dij76. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

DP96. R. Davies and F. Pfenning. A Modal Analysis of Staged Computation. In *23rd Annual ACM Symposium on Principles of Programming Languages.* ACM Press, January 1996.

EJC03. S. Eisenbach, V. Jurisic, and C. Sadler. Modeling the evolution of .NET programs. In *IFIP International Conference on Formal Methods for Open Distributed Systems*, LNCS. Springer Verlag, 2003.

EN94. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, 1994.

FHMV95. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, 1995.

Fit93. M. Fitting. Basic modal logic. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1. Oxford University Press, 1993.

Fit96. M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 2nd edition, 1996.

Fra92. N. Francez. *Program Verification.* Addison-Wesley, 1992.

Fre03. G. Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet.* 1903. Volumes I and II (Jena).

Gal87. J. H. Gallier. *Logic for Computer Science.* John Wiley, 1987.

Gen69. G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, chapter 3, pages 68–129. North-Holland Publishing Company, 1969.

Gol87. R. Goldblatt. *Logics of Time and Computation.* CSLI Lecture Notes, 1987.

Gri82. D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1982.

Ham78. A. G. Hamilton. *Logic for Mathematicians.* Cambridge University Press, 1978.

Hoa69. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

Hod77. W. Hodges. *Logic.* Penguin Books, 1977.

Hod83. W. Hodges. Elementary predicate logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 1. Dordrecht: D. Reidel, 1983.

Hol90. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.

JSS01. D. Jackson, I. Shlyakhter, and M. Sridharan. A Micromodularity Mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC'01)*, September 2001.

Koz83. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

Lee59. C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

Lon83. D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, School of Computer Science, Carnegie Mellon University, July 1983.

Mar01. A. Martin. Adequate sets of temporal connectives in CTL. *Electronic Notes in Theoretical Computer Science* 52(1), 2001.

McM93. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

MP91. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

MP95. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

MvdH95. J.-J. Ch. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science*, volume 41 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1995.

Pap94. C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

Pau91. L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

Pnu81. A. Pnueli. A temporal logic of programs. *Theoretical Computer Science*, 13:45–60, 1981.

Pop94. S. Popkorn. *First Steps in Modal Logic.* Cambridge University Press, 1994.

Pra65. D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

QS81. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, 1981.

Ros97. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

SA91. V. Sperschneider and G. Antoniou. *Logic, A Foundation for Computer Science.* Addison Wesley, 1991.

Sch92. U. Schoening. *Logik für Informatiker.* B. I. Wissenschaftsverlag, 1992.

Sch94. D. A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing. The MIT Press, 1994.

Sim94. A. K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, The University of Edinburgh, Department of Computer Science, 1994.

SS90. G. Stålmarck and M. Såflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems (SAFECOMP'90)*, pages 31–36. Pergamon Press, 1990.

Tay98. R. G. Taylor. *Models of Computation and Formal Languages.* Oxford University Press, 1998.

Ten91. R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

Tur91. R. Turner. *Constructive Foundations for Functional Languages.* McGraw Hill, 1991.

vD89. D. van Dalen. *Logic and Structure.* Universitext. Springer-Verlag, 3rd edition, 1989.

VW84. M. Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. In *Proc. 16th ACM Symposium on Theory of Computing*, pages 446–456, 1984.

Wei98. M. A. Weiss. *Data Structures and Problem Solving Using Java.* Addison-Wesley, 1998.

Index

- ABP, 203
  - acknowledgement channel, 203
  - alternating the control bit, 203
  - fairness, 203
  - main SMV program, 207
- absorption laws, 88
- abstract data type
  - sets, 226
- abstraction, 175, 229, 247
  - and non-determinism, 191
- accessibility relation, 309, 320, 336
- adequate set of connectives
  - for CTL, 216, 222, 231, 397
  - for LTL, 186
  - for propositional logic, 69, 87, 91
- agent, 307, 319, 327
- algebraic specification, 170
- algorithm
  - deterministic, 59
- algorithm `apply`, 373
  - complexity, 380
  - control structure, 374
  - recursive descent, 375
- algorithm `CNF`, 59
- algorithm `reduce`, 372
  - complexity, 380
- algorithm `restrict`, 377
  - complexity, 380
- algorithm `reduce`
  - example execution, 373
- Alloy
  - `[]`, 153
  - `fun`-statement, 155
  - `with`, 146
  - assertion, 144
  - check directive, 144
  - consistency check, 144
  - constrain signature, 150
  - counterexample, 144
  - dot operator, 144
  - extending signature, 169
  - implication, 147
  - let-expression, 153
  - limitations, 156
  - modifiers, 150
  - module, 146
  - opening module, 156
  - polymorphism, 156
  - postcondition, 151
  - precondition, 151
  - reflexive, transitive closure, 169
  - run directive, 146
  - signature, 143
    - instance, 144
  - small scope hypothesis, 143, 168
  - transitive closure, 155
  - universal quantification, 146
- alternating bit protocol, 203
- always in the future, 318
- and-elimination, 6, 339
- and-introduction, 6, 339
- application domain, 173, 257
- approach
  - model-based, 173
  - proof-based, 173
- approximants
  - $\mu_m Z.f$, 392
  - $\nu_m Z.f$, 393
- arity, 99
- array, 299
  - bounds, 136, 287
  - field, 287
  - of integers, 287
  - section, 287
- artificial intelligence, 306
- artificial language, 93
- assertion checking, 143
- assignment, 191
  - initial, 292
  - non-deterministic, 205, 230
  - program notation, 261
  - statement, 261
- associativity laws, 56, 88
- assumption, 4
  - discharging, 28, 329
  - temporary, 11, 121

asynchronous
  circuit, 358
  interleaving, 189
atomic formula, 335
  of modal logic, 307
  of predicate logic
    meaning, 124
axiom
  5, 331
  T, 341
  4, 327, 330, 339
  5, 330, 339
  T, 327, 330, 339, 343
  for assignment, 270
  for equality, 107
  instance, 271

Backus Naur form (BNF), 33
backwards breadth-first search, 225, 232
base case, 41, 42, 86
basic modal logic, 307
BDD, 364
  hi(n), 372
  lo(n), 372
  as boolean function, 365
  complement, 367
  consistent path, 365
  edge, 361
  examples, 365
  has an ordering, 367
  layer of variables, 361
  line
    dashed, 362, 365
    solid, 362, 365
  ordered, 367
  read-1, 405
  reduced, 365
  removal of duplicate non-terminals, 363
  removal of duplicate terminals, 363
  removal of redundant tests, 363
  satisfiable, 365
  subBDD, 363
  which is not a read-1-BDD, 378
  which is not an OBDD, 368
  with duplicated subBDDs, 363
belief, 319
binary decision diagram, 364
binary decision tree, 361
  redundancies in, 362
binding priorities, 176, 209
  for basic modal logic, 307
  for integer expressions, 260
  for $\mathrm{KT45}^n$, 335
  for predicate logic, 101
  for propositional logic, 5
  for relational mu-calculus, 391
bit, 133
  control, 203
  least significant, 381
  most significant, 381
  one-bit channel, 205
  two-bit channel, 205
blocks of code, 261
Boole, G., 91, 374
boolean algebra, 19
boolean connective, 210, 310
boolean existential quantification, 377
boolean expression, 260, 272
boolean forall quantification, 379
boolean formula
  independent of a variable, 375
  semantically equivalent, 374
  truth table, 359
boolean function
  'don't care' conditions, 403
  as a binary decision tree, 361
  symbolic representation, 358
boolean guard, 282
boolean variable, 358
bottom-elimination, 21
bottom-introduction (see 'not-elimination'), 21
box-elimination, 329
box-introduction, 329
branching-time logic, 174

case
  overlap, 61
case analysis, 61, 62, 113
case-statement, 18, 192
characteristic function, 383
Church, A., 133
circuit
  2-bit comparator, 408
  asynchronous, 194, 389
  sequential, 359
  synchronous, 194, 358, 388, 411
circular definition, 217
Clarke, E., 254
classical logic, 30, 328
client, 259
clock tick, 190
CNF, 55
code
  specification, 257
  verification, 257
coding
  AF, 395
  EF, 394
  EG, 395
  EU, 395
  EX, 394
  examples of symbolic evaluation, 395
  fair EG, 398
  fair EU, 397
  fair EX, 397
  set of fair states, 397
command, 261
  atomic, 261
  compound, 261
common knowledge, 332, 335
  as invariant, 390

communicating processes, 256
communication protocol, 194
Compactness Theorem, 137
completeness
  of natural deduction for predicate logic, 96
  of natural deduction for propositional logic, 54
complexity
  exponential, 229
  of `apply`, 404
  of brute force minimal-sum section algorithm, 288
  of fairness, 397
  of labelling algorithm, 224, 225
  of labelling EG_C, 232
composition
  sequential, 278
  synchronous, 194
compositional semantics, 39
compositionality
  in model checking, 230
computability, 131
computation
  intractable, 49
computation path, 180, 212
  fair, 231
computation trace, 285
computation tree logic, 175
computation tree logic, 306, 318
computational behaviour, 306
computer program, 103
concatenation, 126, 132
conclusion, 4, 273
concurrency, 257
conjunct, 56
conjunction, 4, 291
  infinite, 335
connective
  adequate set, 226
  unary, 177, 209
consistency, 308, 316
consistency checking, 143
constant symbol, 157
constraints
  inconsistent, 74
  SAT solver, 69
contradiction, 20, 118, 319, 325
control structure, 261, 262
controlling value, 404
copy rule, 20, 329
core programming language, 259, 287
correspondence theory, 325
counter example, 123, 130, 317, 354
counter trace, 174
critical section, 187
CTL, 175, 254, 306, 318
  as a subset of CTL*, 218
  expressive power, 220
  modalities, 306
  with boolean combinations of path formulas, 220, 251
CTL connectives
  fair, 397
CTL formula
  square brackets, 210
CTL*, 254

DAG, 70
dag, 364
dashed box
  flavour, 339
data structure, 123
de Morgan laws, 57, 216, 251
  for modalities, 313
deadlock, 178, 184, 215, 219
debugging systems, 222
debugging systems, 257
decision problem, 131
  of validity in predicate logic, 133
decision procedure, 55
declarative explanation, 26
declarative sentence, 2, 93
  truth value, 37
default case, 192
definition
  inductive, 33
description
  informal, 258
  language, 172, 174
Dijkstra, E., 283
directed graph, 136, 178, 364
  acyclic, 69, 364
  cycle, 364
disjunction, 4
  of literals, 55, 57
distributivity laws
  of box modality, 314
  of F connective, 185
  of propositional logic, 19, 60, 88
dividend, 302
domain assumption, 148
double negation-elimination, 352
double negation-introduction, 352

elimination rule, 6, 107
Emerson, E. A., 254
encoding, 128
entailment
  in program logics, 278
environment
  and non-determinism, 191
  for concurrent programs, 173
  for predicate logic formulas, 127
equality, 263
  intentional, 107
  program notation, 261
  structural, 153
  symbol, 107
equivalence relation, 321, 327, 339
equivalent formulas
  of basic modal logic, 314
  of CTL, 215–217

- of KT4, 327
- of KT45, 327
- of LTL, 184
- of predicate logic, 117
- of propositional logic, 16
- of relational mu-calculus, 410

exclusive-or, 382, 398
existential quantifier, 216
existential second-order logic, 139, 156
exists-elimination, 113
exists-introduction, 112

factorial
- of a natural number, 262
- program, 262, 284

fairness
- nested fixed points, 398
- symbolic model checking, 396

fairness constraint, 190, 197
- simple, 231, 252

FAIRNESS running, 204
Fibonacci numbers, 85
field index, 287
finite automata, 405
finite data structure, 222
first-order logic, 93
fixed point, 240
- greatest, 240, 241
- least, 240, 241
- semantics for CTL, 217, 238

flow of control, 261
Floyd, R., 269
for-statement, 299
forall-elimination, 109
forall-introduction, 110
formal
- path, 218

formula
- atomic, 175
- height, 44, 86
- Horn, 65
- ill-formed, 177
- immediate subformula, 223
- of basic modal logic, 314
- of CTL, 208
  - atomic, 208
  - ill-formed, 209
  - well-formed, 209
- of LTL
  - valid, 251
- of predicate logic, 100
- of propositional logic, 33, 50
  - well-formed, 32, 33, 44
- of relational mu-calculus, 390
- positive, 328, 343, 348
- scheme, 312, 317
  - **K**, 315
  - in propositional logic, 312
  - instance, 312
- subformula, 35

frame, 322

free for x in ϕ, 106, 109
Frege, G., 170
function
- in predicate logic, 124
- monotone, 240
  - a non-example, 240
- nullary, 99
- recursive, 250
- SAT, 225, 227
  - termination, 253
- SATaf, 228
- SATag, 253
- SATeg, 252
- SATeu, 229
- SATex, 228
- symbol, 96, 98, 157
  - binary, 98
- translate, 250

function $\mathrm{pre}_{\forall}(X)$, 227
function $\mathrm{pre}_{\exists}(X)$, 227, 385
function $\mathrm{pre}_{\forall}(X)$, 385
function SAT
- correctness, 240

future
- excludes the present, 249, 353
- includes the present, 182, 249, 353
- whether it includes the present, 318

G-reachable, 338
- in k steps, 338

Gödel, K., 96
Gentzen, G., 91
Global Assembly Cache, 149
grammar, 33
- clause, 269

guided simulation, 155

Halpern, J., 254
higher-order logic, 141
Hoare triple, 264
Hoare, C. A. R., 264, 269
Hodges, W., 170
Horn clause, 65, 139
hybrid rule, 343

if-statement, 280
implementation
- compliant, 143

implication, 4
- logical, 278

implies-elimination, 9
implies-introduction, 12
in-order representation, 35
inconsistency, 259
index, 132
induction
- course-of-values, 43
- hypothesis, 41, 42
- in model checking, 229
- mathematical, 40

inductive step, 41

infix notation, 125, 210
information
  negative, 343
input parameter, 61
integer
  expression, 260
integer label, 372
integer multiplication, 381
interface between logics, 277
interleaving
  formulas with code, 275
  transitions, 188, 194
introduction rules, 6, 107
introspection
  negative, 319, 326
  positive, 319, 326
intuitionistic logic, 30, 120, 327
invariants, 273
  discovering, 283
  SAT solver, 69
iterative squaring, 412

Jape, 170
justification, 276, 277, 329

Knaster-Tarski Theorem, 241
knowledge
  common, 333
  distributed, 335
  domain-specific, 102
  false, 321
  formula
    positive, 343
  idealised, 319, 327
  in a multi-agent system, 307
  modality, 335
  of agent, 307, 319
Kozen, D., 413
Kripke model, 167, 309
  as a counter example, 354
  for KT45^n, 336
Kripke, S., 309, 315

Löwenheim-Skolem Theorem, 138
label
  adding, 223
  deleting, 224
labelling
  AF, 223
  EG, 224
  EG$_C$, 231
  EU, 223
  EX, 223
labelling algorithm, 222
labelling function
  coding subsets, 383
  for Kripke model, 309
  for LTL model, 178
  frame does not have one, 322
language construct, 299
law of excluded middle, 25

LEM
  instance, 328
linear-time logic, 174
linear-time temporal logic, 175
Linear-time temporal logic (LTL), 175
literal, 55, 62
liveness, 190, 197
  property, 188, 189, 207, 230
logic engineering, 307, 316
logic programming, 49, 170
logical level, 278
logical variables
  of Hoare triple, 269
look-up table, 127, 140
  up-dated, 127
LTL, 175
LTLSPEC, 192, 204

machine state, 263
McMillan, K., 254
memoisation
  of computed OBDDs, 377
midcondition, 270, 276
minimal-sum section, 288
minimal-sum-section problem, 305
modal connective
  C_G, 335
  K_i, 335
modal logic, 306
  K, 326
  KT4, 327
  KT45, 326, 327
  normal, 326
  S4, 327
  S5, 326
modality, 308
  diamond, 308
  path, 220
model
  for propositional logic, 37
  of KT45^n, 336
  of basic modal logic, 309, 324
  of CTL, 310
    pictorial representation, 248, 249
  of intuitionistic propositional logic, 328
  of KT45, 337
  of KT45^n, 337
  of LTL
    pictorial representation, 179
  of predicate logic, 96, 124
  under-specified, 143
model checker, 174
model checking, 141, 173, 175, 256
  algorithm, 217, 225, 231, 318
  debugging, 394
  example, 182, 213
  with fairness constraints, 230
model-based verification, 172, 174
module, 265
modulo 8 counter, 408
modus ponens, 9

modus tollens, 10, 352
muddy-children puzzle, 342, 344
multiplicity constraint, 152
Mutex model
  pictorial representation, 188
mutual exclusion, 187

natural deduction
  extension to predicate logic, 95
  for modal logic, 332
  for temporal logic, 174
  inventor, 91
natural deduction rules
  for basic modal logic, 329
  for KT45^n, 340, 355
  for predicate logic, 107
  for propositional logic, 27
necessity
  logical, 308, 318
  physical, 318
negation, 4
negation-elimination (see 'bottom-elimination'), 21
negation-introduction, 22
nested boolean quantification, 394
network
  synchronous, 174
no strict sequencing, 188, 189, 215
node
  initial, 364
  leaf, 103
  non-terminal, 361
  shared, 70
  terminal, 362, 364
non-blocking protocol, 188, 189, 215
non-determinism, 190
non-termination, 262
normal form, 53, 55
  conjunctive, 55, 360
  disjunctive, 360
  negation, 60
    CTL*, 250
    LTL, 186, 246
  product-of-sums, 406
  sum-of-products, 404
not-elimination, 21
not-introduction, 22

OBDD, 367
  absence of redundant variables, 370
  canonical form, 369
  complementation, 385
  definition, 367
  extensions, 381
  for pre$_\exists(X)$, 387
  for pre$_\forall(X)$, 387
  integer multiplication, 381
  intersection, 385
  limitations, 381
  memoisation, 377
  nested boolean quantification, 380
  of a transition relation, 386
  of an even parity function, 370
  of the odd parity function, 400
  optimal ordering, 406
  reduced, 368
    unique representation, 368
  reduced one for logical 'iff', 400
  representing subsets, 383
  running time of algorithms
    upper bounds, 380
  sensitivity of size, 370
  synthesis of boolean formula, 380
  test
    for implication, 372
    for satisfiability, 372
    for semantic equivalence, 370
    for validity, 372
  union, 385
  variations, 381
odd parity function, 400
omniscience
  logical, 319
or-elimination, 17
or-introduction, 16
overloading
  of $\vDash$, 129
  of proof rules, 107

parity function
  even, 369
    as OBDD, 370
parity OBDD, 381
parse tree
  for a predicate logic formula, 103
  of a term, 159
  of a basic modal logic formula, 307
  of a CTL formula, 210
  of propositional logic formula, 34
  root, 35
  subtree, 35
  underspecified, 312
partial correctness, 265
partial order reduction, 229
pattern
  `checkEU` (f, g), 398
  `checkEX` (f), 397
pattern matching, 6, 111, 279
place holder, 94
possibility, 316
  logical, 308
possible world
  semantics, 315
Post correspondence problem, 132
postcondition, 151
  in program logic, 264
Prawitz, D., 91
precondition, 151
  in program logic, 264
    weakest, 276
  of algorithm, 63

predicate, 93
  binary, 95
  number of arguments, 95
  unary, 95
predicate logic, 93
  consistent set of formulas, 129
  satisfiability, 129
  semantic entailment, 129
  validity, 129
prefix, 126
  notation, 210
  ordering, 126
premise, 270
Prior, A., 254
problem
  instance, 132
  reduction, 132
procedural interpretation, 26
process
  concurrent, 187
  instantiation, 206
processor, 257
program
  behaviour, 264
  bug, 257
  code, 276
  construct, 261
  correctness, 63, 67, 225
  derived, 299
  diverging, 266
  documentation, 257
  environment, 258
  finite-state, 358
  fragment, 262
  logic, 275
  methodology, 259
  procedures, 263
  sequential, 256
  termination, 67, 89, 242, 265
  variable, 227, 268
  verification, 270
    formal, 260
program execution, 316, 319
programming by contract, 296
  Eiffel, 296
programming language
  imperative, 259
proof
  box
    for →i, 11
    for forall-introduction, 110
    for modal logic, 329
    opening, 28
    side by side, 22
  by contradiction, 24
  calculus, 256, 260
  construction, 269
  constructive, 120
  dashed box, 329, 340
  fragment, 278
  indirect, 29
  of correctness, 239
  of termination, 266
  partial, 281
  partial correctness, 269, 300
  search, 49
  solid box, 329
  strategy, 115, 265
  subproof, 272
  tableaux, 269
  theory, 93, 122, 174
  total correctness, 292
proof rules, 5
  for implication, 273
  for assignment, 269
  for conjunction, 6
  for disjunction, 16
  for double negation, 8
  for equality, 108
  for existential quantification, 112
  for if-statements, 272, 280
    modified, 281
  for implication, 12, 277
  for KT45^n, 339
  for negation, 20
  for quantifiers, 112
  for sequential composition, 269, 275
  for universal quantification, 109
  for while-statements, 272, 282, 287
  schema, 111
  subformula property, 113
proof tableaux
  complete, 291
proof-based verification, 172, 256
proposition, 2
propositional logic, 93
protocol, 187, 188
provability
  undecidability of predicate logic, 136

quantifier, 310, 313
  equivalences, 185
  in predicate logic, 94
    binding priorities, 101
    equivalences, 130
    meaning, 123
Quielle, J., 254

reachability, 136, 137
reasoning
  about knowledge, 326, 331
  constructive, 29
  in an arbitrary accessible world, 329
  informal, 343
  quantitative, 259
  unsound, 280
record
  field, 193
recursion
  mutual, 218
recursive call, 280
reductio ad absurdum, 24, 119
reduction to absurdity, 24
reflexive, transitive closure, 167

regular language, 405
relation
  binary, 178
  Euclidean, 321, 327
  functional, 321
  linear, 321
  reflexive, 140, 320, 324
    as formula, 109
  serial, 320, 353
  symmetric, 320
    as formula, 109
  total, 321
  transition, 178
  transitive, 140, 320, 324
    as formula, 109
relational mu-calculus
  fixed-point operators, 392
requirement
  informal, 258, 263, 288
requirements, 142
restriction, 374
right-associative, 5
root of a parse tree, 135
rule
  derived, 23
  hybrid, 10
Russell's paradox, 165

safety property, 187, 189, 207
SAT solver
  cubic, 76
  forcing rules, 71
  permanent marks, 75
  temporary marks, 74
satisfaction
  in a frame, 322
  in a frame for KT45^n, 337
satisfaction relation
  for relational mu-calculus, 391
  for basic modal logic, 310
  for KT45, 337
  for LTL, 180
  for partial correctness, 265
  for predicate logic, 128
  for relational mu-calculus, 391
  for total correctness, 266
satisfiability, 360
  3SAT, 406
  deciding, 65
  of a propositional logic formula, 85
  undecidability of predicate logic, 135
SCC
  fair, 232
scheduler
  fair, 197
scope
  of a dummy variable, 117
  of a variable, 103, 113
  of an assumption, 28, 113, 329
search space, 113, 133
second-order logic, 141
semantic entailment
  for predicate logic, 141
  for basic modal logic, 313
  for KT4, 328
  for normal modal logics, 326
  for predicate logic, 96
  for propositional logic, 46
  for relational mu-calculus, 410
semantic equivalence, 39
semantics
  of $\mu Z.f$, 392
  of $\nu Z.f$, 393
  of basic modal logic, 310
  of boolean quantification, 392
  of CTL, 211
  of EG, 239
  of equality, 131
  of predicate logic, 122
  of propositional logic, 38
  of relational mu-calculus, 391
  of Until, 181
sentence
  atomic, 4
  components, 93
  declarative, 93
  in predicate logic, 128
sequent, 5
  invalid, 116
Shannon expansion, 374
side condition, 108, 110
Sifakis, J., 254
small scope hypothesis, 143
SMV, 254
  main program for ABP, 207
  module, 193
    `receiver`, 205
    `sender`, 204
    for channel, 206
    instantiation, 193
  process, 389
  program
    example, 192
    for Mutex, 195
  specification, 192
software
  life-cycle, 142
  micromodel, 142
  reliability, 149
  requirements, 142
  specification, 142
  validation, 142
soundness
  of forall-elimination, 109
  of natural deduction
    basic modal logic, 354
    predicate logic, 96, 122
    propositional logic, 45
  of program logics, 267
  of proof rule for while-statements, 282
  of the substitution principle, 108

specification
  for ABP, 207
  formal, 259
  informal, 259
  language, 172
  of a predicate, 157
  patterns, 254
  practical pattern, 183, 215
  truth table, 58
specifications, 191
Spin, 254
state
  critical, 188
  explosion, 229
  explosion problem, 254
  fair, 397
  final, 142
  formula, 218
  global, 188
  graph, 180
  initial, 142, 189, 222, 247, 252, 264
  non-critical, 188
  of a system, 269
  of core program, 264
  reachable, 247
  resulting, 263, 299
  space, 229
  splitting states, 190
  transition, 142
  trying, 188
state machine, 142
storage
  location, 288
  state, 261
store
  of core program, 264
string, 247, 307
  binary, 126, 132
  empty, 126
strongly connected component, 225
structural equality, 153
structural induction, 44, 51
subformula, 178
substitution
  in predicate logic, 105
  instance, 323
  instance of tautology, 314
  principle, 108
symbolic model checking, 383
syntactic
  domain, 260, 261
syntax
  of basic modal logic, 307
  of boolean expressions, 261
  of boolean formulas, 398
  of CTL, 208
  of CTL*, 218
  of KT45^n, 335
  of LTL, 175
  of predicate logic, 100
  of propositional logic, 33
  of relational mu-calculus, 390
  of terms, 99
system
  asynchronous, 254
    interleaving model, 389
    simultaneous model, 389
  axiomatic, 91
  commercially critical, 172, 257
  component, 206
  concurrent, 173
  debugging, 174
  description, 193
  design, 174
  development, 173
  elevator, 184, 215
  finite-state, 256
  hybrid, 277
  infinite-state, 256
  mission-critical, 172
  multi-agent, 331
  physical, 175
  reactive, 173, 257, 358
  safety-critical, 172, 257
  transition, 174
  verification, 256

tautology, 50
temporal connective
  AF, 212
  AG, 211
  AU, 212
  AX, 211
  EF, 212
  EG, 211
  EU, 212
  EX, 211
temporal connectives, 176
temporal logic, 174, 306
term, 99
  interpretation, 128
term-rewriting system, 170
termination
  Collatz $3n + 1$, 295
  proof, 266
tertium non datur, 25
theorem, 13
  prover, 106, 136
  proving, 170
time
  continuous, 174
  discrete, 174
top
  marking, 66
total correctness, 265, 266
transition relation, 178
  for SMV programs, 388
transition system, 174
  of ABP program, 247
  of Mutex code, 198
  of SMV program, 192
  unwinding, 180, 212, 222

translation
  English into predicate logic, 95, 101
tree
  infinite, 180, 212
truth
  dynamic, 174
  mode, 306, 308
  of knowledge, 326
  static, 174
  value
    for predicate logic, 127
    for propositional logic, 3
truth table
  for conjunction, 37
truth tables, 38
type, 12, 327
  checking, 12
  theory, 170

unary connective, 307
undecidability
  of provability, 136
  of satisfiability, 135
  of validity in predicate logic, 133
universal quantification, 268
universal quantifier, 216
universal second-order logic, 140, 156
universe of concrete values, 124
unreachability, 140
unsound sequent, 164
Until
  in natural language, 182
  negating, 187
updated valuation, 391

valid sequent
  of modal logic, 330
  partial correctness, 267
  total correctness, 267
validity
  in basic modal logic, 314
  in KT45^n, 339
  in propositional logic, 85
  undecidability in predicate logic, 133
valuation
  for propositional logic, 37
  in predicate logic, 123
  in relational mu-calculus, 391
value
  initial, 206, 268, 269
Vardi, M., 254
variable, 94, 260
  boolean, 229, 247, 358
  bound, 103
  capture, 106
  dummy, 110
  free, 103
  local, 263
  logical, 268, 290
variable ordering
  compatible, 368
  list, 367
variant, 293
verification
  full, 173
  method, 172
  of communication protocols, 175
  of hardware, 175
  of software, 175
  of systems, 256
  post-development, 173, 257
  pre-development, 173, 257
  process, 271
  program, 270
  property, 173
  property-oriented, 256
  semi-automatic, 256
  techniques, 172

weakest precondition, 276
while-statement, 261, 262
  body, 273, 282, 286
  non-termination, 292
wise-men puzzle, 342
Wolper, P., 254
word
  empty, 126
world
  accessible, 309
  possible, 309, 336

year-2000 problem, 258