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.

Contents

Foreword to the first edition — *ix*

Preface to the second edition — *xi*

Acknowledgments — *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 . . . 24
  - 1.2.3 Natural deduction in summary . . . 26
  - 1.2.4 Provable equivalence . . . 29
  - 1.2.5 An aside: proof by contradiction . . . 30
- 1.3 Propositional logic as a formal language . . . 32
- 1.4 Semantics of propositional logic . . . 37
  - 1.4.1 The meaning of logical connectives . . . 37
  - 1.4.2 Mathematical induction . . . 41
  - 1.4.3 Soundness of propositional logic . . . 46
  - 1.4.4 Completeness of propositional logic . . . 50
- 1.5 Normal forms . . . 55
  - 1.5.1 Semantic equivalence, satisfiability and validity . . . 55
  - 1.5.2 Conjunctive normal forms and validity . . . 60
  - 1.5.3 Horn clauses and satisfiability . . . 67
- 1.6 SAT solvers . . . 70
  - 1.6.1 A linear solver . . . 71

1.6.2 A cubic solver 74
1.7 Exercises 80
1.8 Bibliographic notes 96

2 Predicate logic ***97***
2.1 The need for a richer language 97
2.2 Predicate logic as a formal language 102
2.2.1 Terms 103
2.2.2 Formulas 104
2.2.3 Free and bound variables 106
2.2.4 Substitution 108
2.3 Proof theory of predicate logic 111
2.3.1 Natural deduction rules 111
2.3.2 Quantifier equivalences 121
2.4 Semantics of predicate logic 127
2.4.1 Models 128
2.4.2 Semantic entailment 134
2.4.3 The semantics of equality 136
2.5 Undecidability of predicate logic 136
2.6 Expressiveness of predicate logic 141
2.6.1 Existential second-order logic 144
2.6.2 Universal second-order logic 145
2.7 Micromodels of software 147
2.7.1 State machines 147
2.7.2 Alma — re-visited 152
2.7.3 A software micromodel 153
2.8 Exercises 162
2.9 Bibliographic notes 178

3 Verification by model checking ***180***
3.1 Motivation for verification 180
3.2 Linear-time temporal logic 183
3.2.1 Syntax of LTL 184
3.2.2 Semantics of LTL 185
3.2.3 Practical patterns of specifications 192
3.2.4 Important equivalences between LTL formulas 193
3.2.5 Adequate sets of connectives for LTL 194
3.3 Model checking: systems, tools, properties 196
3.3.1 Example: mutual exclusion 196

3.3.2 The NuSMV model checker 200
3.3.3 Running NuSMV 203
3.3.4 Mutual exclusion revisited 204
3.3.5 The ferry-man 207
3.3.6 The alternating bit protocol 212
3.4 Branching-time logic 216
3.4.1 Syntax of CTL 216
3.4.2 Semantics of computation tree logic 219
3.4.3 Practical patterns of specifications 223
3.4.4 Important equivalences between CTL formulas 224
3.4.5 Adequate sets of CTL connectives 225
3.5 CTL* and the expressive powers of LTL and CTL 226
3.5.1 Boolean combinations of temporal formulas in CTL . 229
3.5.2 Past operators in LTL 230
3.6 Model checking algorithms 231
3.6.1 The CTL model-checking algorithm 231
3.6.2 CTL model checking with fairness 239
3.6.3 The LTL model checking algorithm 241
3.7 The fixed-point characterisation of CTL 247
3.7.1 Monotone functions 249
3.7.2 The correctness of $\mathtt{SAT}_{\mathtt{EG}}$ 251
3.7.3 The correctness of $\mathtt{SAT}_{\mathtt{EU}}$ 252
3.8 Exercises 254
3.9 Bibliographic notes 264

4 Program verification ***266***
4.1 Why should we specify and verify code? 267
4.2 A framework for software verification 268
4.2.1 A core programming language 270
4.2.2 Hoare triples 273
4.2.3 Partial and total correctness 276
4.2.4 Program variables and logical variables 278
4.3 Proof calculus for partial correctness 280
4.3.1 Proof rules 280
4.3.2 Proof tableaux 284
4.3.3 A case study: minimal-sum section 299
4.4 Proof calculus for total correctness 304
4.5 Programming by contract 308
4.6 Exercises 311

4.7 Bibliographic notes 318

5 Modal logics and agents ***319***

5.1 Modes of truth 319
5.2 Basic modal logic 320
5.2.1 Syntax 320
5.2.2 Semantics 321
5.3 Logic engineering 328
5.3.1 The stock of valid formulas 330
5.3.2 Important properties of the accessibility relation . . . 333
5.3.3 Correspondence theory 335
5.3.4 Some modal logics 340
5.4 Natural deduction 343
5.5 Reasoning about knowledge in a multi-agent system 346
5.5.1 Some examples 346
5.5.2 The modal logic $\mathrm{KT45}^n$ 349
5.5.3 Natural deduction for $\mathrm{KT45}^n$ 354
5.5.4 Formalising the examples 357
5.6 Exercises 364
5.7 Bibliographic notes 372

6 Binary decision diagrams ***374***

6.1 Representing boolean functions 374
6.1.1 Propositional formulas and truth tables 375
6.1.2 Binary decision diagrams 377
6.1.3 Ordered BDDs 383
6.2 Algorithms for reduced OBDDs 388
6.2.1 The algorithm `reduce` 388
6.2.2 The algorithm `apply` 389
6.2.3 The algorithm `restrict` 392
6.2.4 The algorithm `exists` 393
6.2.5 Assessment of OBDDs 397
6.3 Symbolic model checking 399
6.3.1 Representing subsets of the set of states 400
6.3.2 Representing the transition relation 402
6.3.3 Implementing the functions $\mathrm{pre}_\exists$ and $\mathrm{pre}_\forall$ 404
6.3.4 Synthesising OBDDs 404
6.4 A relational mu-calculus 407
6.4.1 Syntax and semantics 407

6.4.2 Coding CTL models and specifications 410
6.5 Exercises 415
6.6 Bibliographic notes 432

Bibliography ***433***

Index ***437***

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`

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[1].

$\wedge$: 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[2]. 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

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.

1 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.

2 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.

sentence has the structure 'p is the case **and if** q **then** ... ' A computer would require the insertion of brackets, as in

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

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 $\lor$ and $\land$, and the latter two bind more tightly than $\to$. Implication $\to$ is *right-associative*: expressions of the form $p \to q \to r$ denote $p \to (q \to r)$.

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[1] 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 $p \land \neg q \to r, \neg r, p \vdash q$. 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

[1] 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		

conclusion. Additionally, our proof rules should be carefully chosen; otherwise, we might be able to 'prove' invalid patterns of argumentation. For 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:

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

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:

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

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

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

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

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

The rule for eliminating implication

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

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

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

p: It rained.
$p \rightarrow q$: 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 $\rightarrow$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.
$p \rightarrow q$: 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 $p \rightarrow q$, but if it doesn't satisfy p — e.g. if its input is a surname — then we will not be able to derive q.

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

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

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

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

Before turning to implies-introduction, let's look at a hybrid rule which has the Latin name *modus tollens.* It is like the $\to$e rule in that it eliminates an implication. Suppose that $p \to q$ *and* $\neg q$ are the case. Then, *if* p holds we can use $\to$e to conclude that q holds. Thus, we then have that q *and* $\neg q$ hold, which is impossible. Therefore, we may infer that p must be false. But this can only mean that $\neg p$ is true. We summarise this reasoning into the rule *modus tollens*, or MT for short:[1]

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

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

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

we use several of the rules introduced so far:

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

Examples 1.8 Here are two example proofs which combine the rule MT

[1] 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.

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 $\to$i:

$$\begin{array}{r|ll|}
\multicolumn{1}{r}{1} & \neg q \to \neg p & \multicolumn{1}{l}{\text{premise}} \\ \cline{2-3}
2 & p & \text{assumption} \\
3 & \neg\neg p & \neg\neg\text{i } 2 \\
4 & \neg\neg q & \text{MT } 1,3 \\ \cline{2-3}
\multicolumn{1}{r}{5} & p \to \neg\neg q & \multicolumn{1}{l}{\to\text{i } 2\text{--}4}
\end{array}$$

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

$$\begin{array}{rll}
1 & p & \text{premise}
\end{array}$$

which demonstrates $p \vdash p$. The rule $\to$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

$$\begin{array}{r|ll|}
\cline{2-3}
1 & p & \text{assumption} \\ \cline{2-3}
\multicolumn{1}{r}{2} & p \to p & \multicolumn{1}{l}{\to\text{i } 1-1}
\end{array}$$

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	$\to$e 1, 6
8	$p \to r$	$\to$i 3–7
9	$(\neg q \to \neg p) \to (p \to r)$	$\to$i 2–8
10	$(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$	$\to$i 1–9

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

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

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

by 'augmenting' the previous proof with n lines of the rule $\to$i applied to ϕ_n, ϕ_{n-1}, $\ldots$, 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 $\to$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;

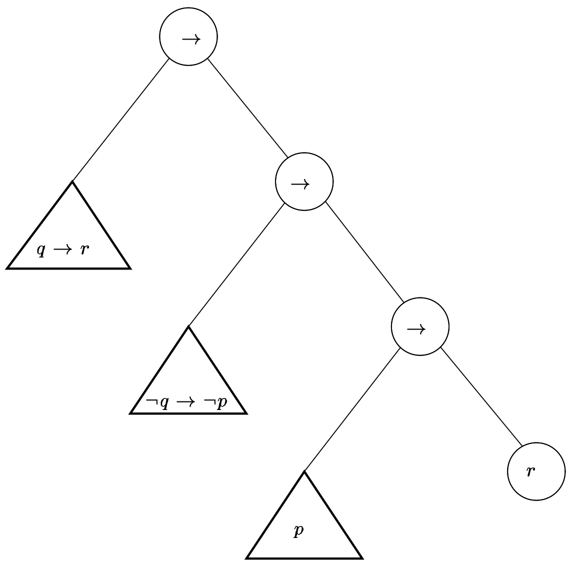


Fig. 1.1. Part of the structure of the formula $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ to show how it determines the proof structure.

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 $\to$i. The formula $p \to r$ in line 8 is yet another implication. Therefore, we have to assume p in line 3 and hope to show r in line 7, then $\to$i produces the desired result in line 8.

The remaining question now is this: how can we show r, using the three assumptions in lines 1–3? This, and only this, is the creative part of this proof. We see the implication $q \to r$ in line 1 and know how to get r (using $\to$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 $\neg\neg$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 $\neg\neg$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 $\wedge$i, we can prove the validity of the sequent

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

1	$p \wedge q \to r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	$\wedge$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 $2{-}6$

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	$\wedge e_1$ 2
4	r	$\wedge e_2$ 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 $\vee i_1$ and $\vee i_2$. 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

$$\frac{\phi}{\phi \vee \psi}\ \vee i_1 \qquad\qquad \frac{\psi}{\phi \vee \psi}\ \vee i_2\ .$$

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	$q \vee p$	$\vee$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 $q \rightarrow r \vdash p \vee q \rightarrow p \vee r$ is valid:

1	$q \to r$	premise
2	$p \vee q$	assumption
3	p	assumption
4	$p \vee r$	$\vee$i$_1$ 3
5	q	assumption
6	r	$\to$e $1, 5$
7	$p \vee r$	$\vee$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, $\vee$i$_1$ and $\vee$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	$p \vee (q \vee r)$	$\vee$i$_1$ 3
5	q	assumption
6	$q \vee r$	$\vee$i$_1$ 5
7	$p \vee (q \vee r)$	$\vee$i$_2$ 6
8	$p \vee (q \vee r)$	$\vee$e $2, 3{-}4, 5{-}7$
9	r	assumption
10	$q \vee r$	$\vee$i$_2$ 9
11	$p \vee (q \vee r)$	$\vee$i$_2$ 10
12	$p \vee (q \vee r)$	$\vee$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	$p \land (q \lor r)$	premise
2	p	$\land e_1$ 1
3	$(q \lor r)$	$\land e_2$ 1
4	q	assumption
5	$(p \land q)$	$\land$i 2, 4
6	$(p \land q) \lor (p \land r)$	$\lor i_1$ 5
7	r	assumption
8	$(p \land r)$	$\land$i 2, 7
9	$(p \land q) \lor (p \land r)$	$\lor i_2$ 8
10	$(p \land q) \lor (p \land r)$	$\lor$e 3, 4–6, 7–9

verifies the validity of the sequent $p \land (q \lor r) \vdash (p \land q) \lor (p \land r)$ and you are encouraged to show the validity of the 'converse' $(p \land q) \lor (p \land r) \vdash p \land (q \lor r)$ 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: The moon is made of green cheese.
q: 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 $\neg p \vee q \vdash p \to q$ 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	$p \to q$	$\to$i $3{-}4$
6	$p \to q$	$\to$i $3{-}5$		
7	$p \to q$			$\vee$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:

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

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 ¬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

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

from $\to$e, $\neg$e and $\neg$i:

$$\begin{array}{rll}
1 & \phi \to \psi & \text{premise} \\
2 & \neg\psi & \text{premise} \\ \cline{2-3}
3 & \multicolumn{1}{|l}{\phi} & \multicolumn{1}{l|}{\text{assumption}} \\
4 & \multicolumn{1}{|l}{\psi} & \multicolumn{1}{l|}{\to\text{e } 1,3} \\
5 & \multicolumn{1}{|l}{\bot} & \multicolumn{1}{l|}{\neg\text{e } 4,2} \\ \cline{2-3}
6 & \neg\phi & \neg\text{i } 3\text{--}5
\end{array}$$

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

The same holds for the rule

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

It can be derived from the rules $\neg$i and $\neg$e, as follows:

$$\begin{array}{rll}
1 & \phi & \text{premise} \\ \cline{2-3}
2 & \multicolumn{1}{|l}{\neg\phi} & \multicolumn{1}{l|}{\text{assumption}} \\
3 & \multicolumn{1}{|l}{\bot} & \multicolumn{1}{l|}{\neg\text{e } 1,2} \\ \cline{2-3}
4 & \neg\neg\phi & \neg\text{i } 2\text{--}3
\end{array}$$

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:

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

This rule looks rather similar to $\neg$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 $\rightarrow$i, we can transform this into a proof of $\neg\phi \rightarrow \bot$ and proceed as follows:

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

This shows that PBC can be derived from $\rightarrow$i, $\neg$i, $\rightarrow$e and $\neg\neg$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 $\phi \vee \neg\phi$ 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 $\vdash \phi \vee \neg\phi$ 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 $B \vee \neg B$ 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	$\neg(\phi \vee \neg\phi)$	assumption
2	ϕ	assumption
3	$\phi \vee \neg\phi$	$\vee\mathrm{i}_1$ 2
4	$\bot$	$\neg$e 3, 1
5	$\neg\phi$	$\neg$i 2–4
6	$\phi \vee \neg\phi$	$\vee\mathrm{i}_2$ 5
7	$\bot$	$\neg$e 6, 1
8	$\neg\neg(\phi \vee \neg\phi)$	$\neg$i 1–7
9	$\phi \vee \neg\phi$	$\neg\neg$e 8

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

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

- $\wedge$i says: to prove $\phi \wedge \psi$, you must first prove ϕ and ψ separately and then use the rule $\wedge$i.

The basic rules of natural deduction:

	introduction	*elimination*
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\ \wedge\mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\ \wedge\mathrm{e}_1$ $\quad \dfrac{\phi \wedge \psi}{\psi}\ \wedge\mathrm{e}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\ \vee\mathrm{i}_1$ $\quad \dfrac{\psi}{\phi \vee \psi}\ \vee\mathrm{i}_2$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ \vee\mathrm{e}$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \rightarrow \psi}\ \rightarrow\mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\ \rightarrow\mathrm{e}$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}\ \neg\mathrm{i}$	$\dfrac{\phi \quad \neg\phi}{\bot}\ \neg\mathrm{e}$
$\bot$	(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:

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

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

Fig. 1.2. Natural deduction rules for propositional logic.

- $\wedge e_1$ says: to prove ϕ, try proving $\phi \wedge \psi$ and then use the rule $\wedge e_1$. Actually, this doesn't sound like very good advice because probably proving $\phi \wedge \psi$ will be harder than proving ϕ alone. However, you might find that you *already have* $\phi \wedge \psi$ lying around, so that's when this rule is useful. Compare this with the example sequent in Example 1.15.
- $\vee i_1$ says: to prove $\phi \vee \psi$, try proving ϕ. Again, in general it is harder to prove ϕ than it is to prove $\phi \vee \psi$, so this will usually be useful only if you've already managed to prove ϕ. For example, if you want to prove $q \vdash p \vee q$, you certainly won't be able simply to use the rule $\vee i_1$, but $\vee i_2$ will work.
- $\vee e$ has an excellent procedural interpretation. It says: if you have $\phi \vee \psi$, 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, $\rightarrow i$ says, if you want to prove $\phi \rightarrow \psi$, 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).

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 $\phi \rightarrow \psi$, then apply [1] the rule $\rightarrow i$. This means drawing a box with ϕ at the top and ψ at the bottom.

[1] Except in situations such as $p \rightarrow (q \rightarrow \neg r), p \vdash q \rightarrow \neg r$ where $\rightarrow e$ produces a simpler proof

So your proof, which started out like this:

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

now looks like this:

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

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.

The proof rule $\neg$i is very similar to $\to$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 $\to$i and $\neg$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 $\vdash (\phi \to \psi) \wedge (\psi \to \phi)$ is valid; it defines the same concept. Examples of provably equivalent formulas are

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

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

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

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

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

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 $\neg\neg$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 $\rightarrow$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 $\{p, q, r, \ldots\} \cup \{p_1, p_2, p_3, \ldots\} \cup \{\neg, \wedge, \vee, \to, (,)\}$. This is a trivial observation and as such is not good enough for what we are trying to capture. For example, the string $(\neg)()\vee pq \to$ 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 $p, q, r, \ldots$ and $p_1, p_2, p_3, \ldots$ 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 $(\phi \vee \psi)$.
$\to$: If ϕ and ψ are well-formed formulas, then so is $(\phi \to \psi)$.

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

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

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

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

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 premise $((\neg p) \wedge q)$ and conclusion $(p \wedge (q \vee (\neg r)))$. By applying the inversion principle to the premise, 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 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[1] 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 $(\neg)() \vee pq \to$ from above. We can decide this matter by being very observant. The string $(\neg)() \vee pq \to$ contains $\neg)$

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

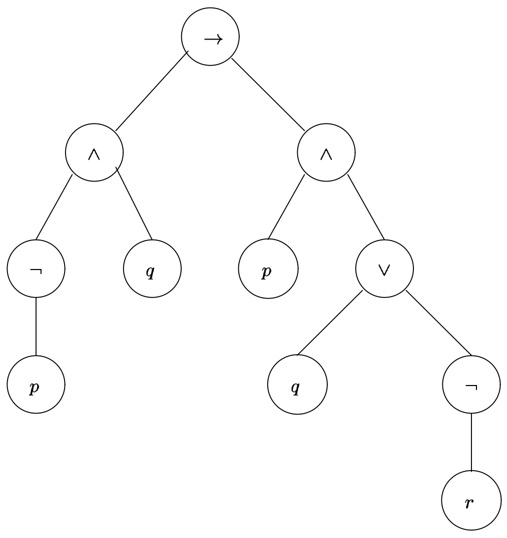


Fig. 1.3. A parse tree representing a well-formed formula.

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, $(\neg)() \vee pq \to$ 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 formula in (1.4) is well-formed. In Figure 1.3 we see that its parse tree has $\to$ 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 $\to$. In the case of $\neg$ there is only *one* subtree coming out of the root. In the cases $\wedge$, $\vee$ or $\to$ 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 $\to$

and $(\neg r)$, $(q \lor (\neg r))$ and $((p \land (q \lor (\neg p))))$ 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

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

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, $\land$, $\lor$ 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[1]. The resulting well-formed formula is $((\neg(p \lor (q \rightarrow (\neg p)))) \land r)$.

The tree in Figure 1.21 on page 85, however, does *not* represent a well-formed formula for two reasons. First, the leaf $\land$ (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] 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.

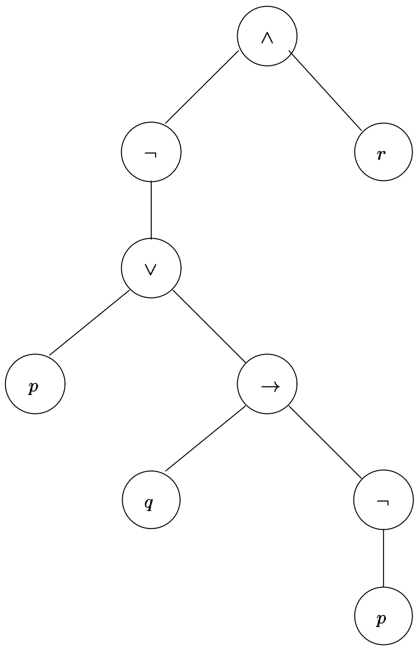


Fig. 1.4. Given: a tree; wanted: its linear representation as a logical formula.

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

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

Fig. 1.5. The truth table for conjunction, the logical connective $\wedge$.

$\wedge$, then the truth value of $p \wedge q$ is determined by three things: the truth value of p, the truth value of q and the meaning of $\wedge$. The meaning of $\wedge$ is captured by the observation that $p \wedge q$ is true iff p *and* q are both true; otherwise $p \wedge q$ is false. Thus, as far as $\wedge$ 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 $p \vee \neg q$. Please list the remaining three valuations for this formula.

We can think of the meaning of $\wedge$ 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 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

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	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

ϕ	$\neg\phi$
T	F
F	T

$\top$
T

$\bot$
F

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

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 $\rightarrow$ as checking whether *truth is being preserved.* Clearly, this is not the case when we have T $\rightarrow$ F, since we infer something that is false from something that is true. So the second entry in the column $\phi \rightarrow \psi$ equals F. On the other hand, T $\rightarrow$ T obviously preserves truth, but so do the cases F $\rightarrow$ T and F $\rightarrow$ 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 $\rightarrow$, then it might be good to think of $\phi \rightarrow \psi$ 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 $\phi \rightarrow \psi$ 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 $\neg p \wedge q$

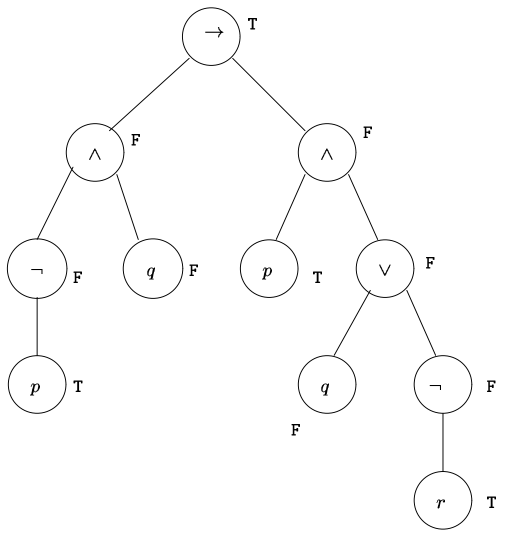


Fig. 1.7. The evaluation of a logical formula under a given valuation.

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	q	$\neg p$	$\neg q$	$p \to \neg q$	$q \lor \neg p$	$(p \to \neg q) \to (q \lor \neg p)$
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

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

$p \to \neg q$ accordingly. For example, in the first line p is T and $\neg q$ is F so the entry for $p \to \neg q$ is $\mathtt{T} \to \mathtt{F} = \mathtt{F}$ by definition of the meaning of $\to$. 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 $\lor$ with columns 2 and 3 as input. Finally, column 7 results from applying the truth table of $\to$ 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.[1] 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] 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.

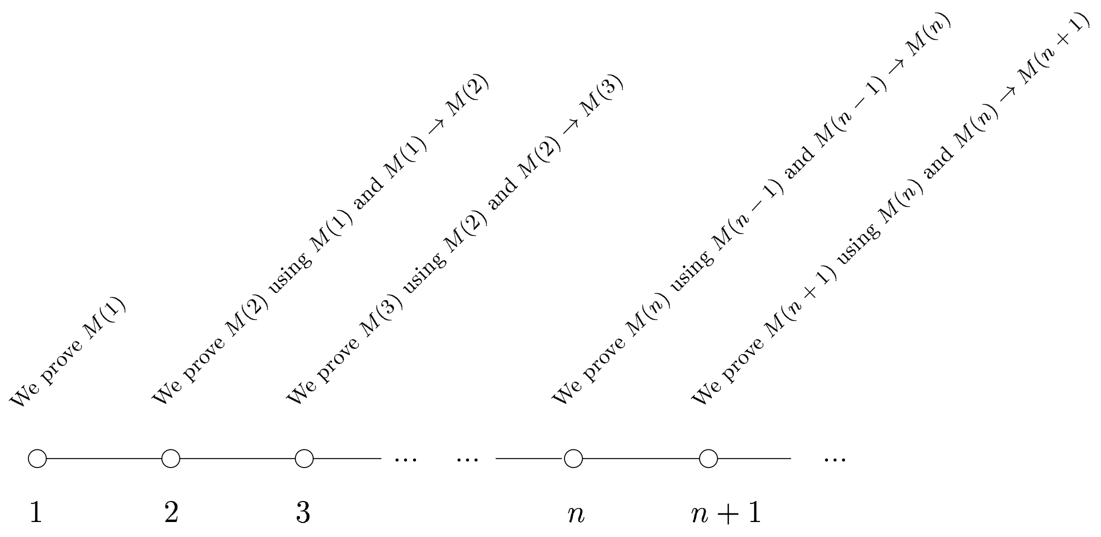

We prove $M(1)$
We prove $M(2)$ using $M(1)$ and $M(1) \to M(2)$
We prove $M(3)$ using $M(2)$ and $M(2) \to M(3)$
We prove $M(n)$ using $M(n-1)$ and $M(n-1) \to M(n)$
We prove $M(n+1)$ using $M(n)$ and $M(n) \to M(n+1)$
1
2
3
n
$n+1$

Fig. 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.

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.

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

$$\begin{aligned}
&\mathrm{LHS}_{n+1}\\
&\quad = 1 + 2 + 3 + 4 + \cdots + (n + 1)\\
&\quad = \mathrm{LHS}_n + (n + 1) \quad \text{regrouping the sum}\\
&\quad = \mathrm{RHS}_n + (n + 1) \quad \text{by our induction hypothesis}\\
&\quad = \tfrac{n\cdot(n+1)}{2} + (n + 1)\\
&\quad = \tfrac{n\cdot(n+1)}{2} + \tfrac{2\cdot(n+1)}{2} \quad \text{arithmetic}\\
&\quad = \tfrac{(n+2)\cdot(n+1)}{2} \quad \text{arithmetic}\\
&\quad = \tfrac{((n+1)+1)\cdot(n+1)}{2} \quad \text{arithmetic}\\
&\quad = \mathrm{RHS}_{n+1}\ .
\end{aligned}$$

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 $M(1) \wedge M(2) \wedge M(3) \rightarrow M(4)$. Now consider $n = 1$: the inductive-step instance is $M(1) \rightarrow M(2)$. What about the case when n equals 0? In this case, there are zero formulas on the left of the $\rightarrow$, 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.

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.

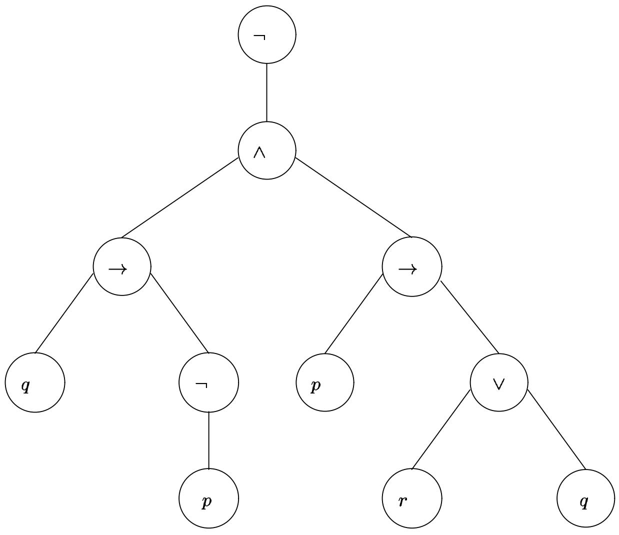


Fig. 1.10. A parse tree with height 5.

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 $(p \to (q \land \neg r))$ 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 T, ψ evaluates to T as well.

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

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

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

Let us look at some examples of this notion.

1. Does $p \land q \vDash p$ 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 T for $p \land q$ we need to make sure that p is true as well. But $p \land q$ computes T only if p and q are true, so $p \land q \vDash p$ is indeed the case.
2. What about the relationship $p \lor q \vDash p$? There are three assignments for which $p \lor q$ computes T, so p would have to be true for all of these. However, if we assign T to q and F to p, then $p \lor q$ computes T, but p is false. Thus, $p \lor q \vDash p$ 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 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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ 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:

1	ϕ_1	premise
2	ϕ_2	premise
	$\vdots$	
n	ϕ_n	premise
	$\vdots$	
k	ψ	justification

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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_1$ and $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_2$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_1$ and $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_2$ holds. But these two relations imply that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_1 \wedge \psi_2$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \eta_1 \vee \eta_2$ 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 $\phi_1, \phi_2, \ldots, \phi_n, \eta_1 \vdash \psi$ and $\phi_1, \phi_2, \ldots, \phi_n, \eta_2 \vdash \psi$ from the case analysis of $\vee$e. By our induction hypothesis, we conclude that the relations $\phi_1, \phi_2, \ldots, \phi_n \vDash \eta_1 \vee \eta_2$, $\phi_1, \phi_2, \ldots, \phi_n, \eta_1 \vDash \psi$ and $\phi_1, \phi_2, \ldots, \phi_n, \eta_2 \vDash \psi$ hold. But together these three relations then force that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ 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.

$\square$

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 $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ 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 $\phi_1, \phi_2, \ldots, \phi_2 \vDash \psi$. Using soundness, this means that $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ 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

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

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.

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, $\phi_2, \dots, \phi_n$ 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.

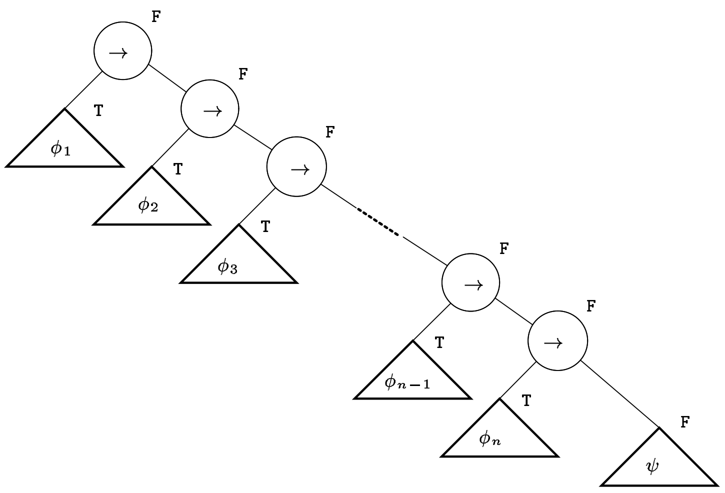


Fig. 1.11. The only way this parse tree can evaluate to F. We represent parse trees for ϕ_1, ϕ_2, ... , ϕ_n as triangles as their internal structure does not concern us here.

Step 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 $1 \leq i \leq n$ 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 $\mathtt{T}$. In this case ϕ_1 evaluates to $\mathtt{F}$. 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 $\mathtt{F}$, then ϕ_1 evaluates to $\mathtt{T}$ 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 $\to$, $\wedge$ 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 $\phi_1 \to \phi_2$. If ϕ evaluates to $\mathtt{F}$, then we know that ϕ_1 evaluates to $\mathtt{T}$ and ϕ_2 to $\mathtt{F}$. 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \neg(\phi_1 \to \phi_2)$; but using $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$, this amounts to proving the sequent $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \to \phi_2)$, which we leave as an exercise.

   If ϕ evaluates to $\mathtt{T}$, then we have three cases. First, if ϕ_1 evaluates to $\mathtt{F}$ and ϕ_2 to $\mathtt{F}$, 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 $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \to \phi_2$, which we leave as an exercise. Second, if ϕ_1 evaluates to $\mathtt{F}$ and ϕ_2 to $\mathtt{T}$, we use our induction hypothesis to arrive at $\hat{p}_1, \dots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ and have to prove $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to $\mathtt{T}$, we arrive at $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, using our induction hypothesis, and need to prove $\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$, 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 $\hat{q}_1, \dots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \dots, \hat{r}_k \vdash \phi_2$ by our induction hypothesis, so $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ follows. Second, if ϕ_1 evaluates to F and ϕ_2 to T, then we get $\hat{p}_1, \dots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$; 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$, 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 $\hat{p}_1, \dots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ 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.

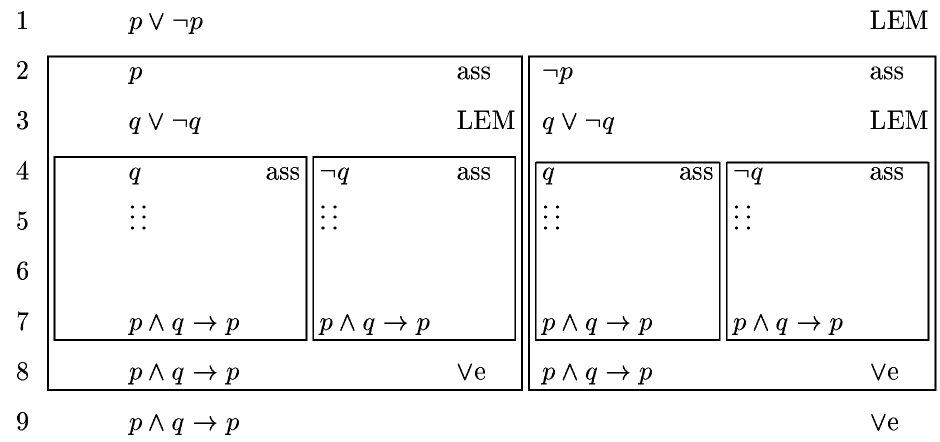
$\square$

We apply this technique to the formula $\vDash \phi_1 \rightarrow (\phi_2 \rightarrow (\phi_3 \rightarrow (\dots (\phi_n \rightarrow \psi) \dots)))$. 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 $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \eta$, 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 $p \wedge q \to p$ 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 $r \vee \neg r$, 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 $\vee$e. That way we can invoke all four proofs of the sequents above and use the rule $\vee$e repeatedly until we have got rid of all our premises. We spell out the combination of these four phases schematically:



1 $p \vee \neg p$ LEM
2 p ass $\neg p$ ass
3 $q \vee \neg q$ LEM $q \vee \neg q$ LEM
4 q ass $\neg q$ ass q ass $\neg q$ ass
5 $\vdots$ $\vdots$ $\vdots$ $\vdots$
6
7 $p \wedge q \to p$ $p \wedge q \to p$ $p \wedge q \to p$ $p \wedge q \to p$
8 $p \wedge q \to p$ $\vee$e $p \wedge q \to p$ $\vee$e
9 $p \wedge q \to p$ $\vee$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 $p \wedge q \to p$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$. Take the proof for $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$ given by step 2 and augment its proof by introducing $\phi_1, \phi_2, \ldots, \phi_n$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$.

Corollary 1.39 (Soundness and Completeness) *Let $\phi_1, \phi_2, \ldots, \phi_n, \psi$ be formulas of propositional logic. Then $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ is holds iff the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ 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 applied this to show that a sequent does not have a proof: simply show that $\phi_1, \phi_2, \dots, \phi_2$ does not semantically entail ψ; then soundness implies that the sequent $\phi_1, \phi_2, \dots, \phi_2 \vdash \psi$ 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 $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds. We now investigate various alternatives for deciding $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ 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 $\vDash (\phi \to \psi) \land (\psi \to \phi)$ 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 $\phi_1, \phi_2, \ldots, \phi_n$ and ψ of propositional logic, $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds iff $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ holds.*

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

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 :

$$\begin{array}{rcl} L & ::= & p \mid \neg p \\ D & ::= & L \mid L \vee D \\ C & ::= & D \mid D \wedge C. \end{array} \tag{1.6}$$

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

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

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 41). 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 $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$.

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 $\neg\phi_1 \vee \neg\phi_2 \vee \cdots \vee \neg\phi_n \equiv \neg(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$, 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:

$$\begin{array}{llll} \psi_1 \stackrel{\text{def}}{=} \neg p \vee \neg q \vee r & \text{(line 2)} & \psi_2 \stackrel{\text{def}}{=} p \vee \neg q \vee \neg r & \text{(line 5)} \\ \psi_3 \stackrel{\text{def}}{=} p \vee \neg q \vee r & \text{etc} & \psi_4 \stackrel{\text{def}}{=} p \vee q \vee \neg r\ . & \end{array}$$

The resulting ϕ in CNF is therefore

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

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 $\to$ 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 $\phi_1 \land \phi_2$, we call `CNF` recursively on each ϕ_i to get the respective output η_i and return the CNF $\eta_1 \land \eta_2$ as output for input ϕ.
- If ϕ equals $\phi_1 \lor \phi_2$, we again call `CNF` recursively on each ϕ_i to get the respective output η_i; but this time we must not simply return $\eta_1 \lor \eta_2$ 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 $\phi_1 \lor \phi_2$. 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 $\rightarrow$ 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 $\phi_1 \land \phi_2$:  return NNF ($\phi_1$) $\land$ NNF ($\phi_2$)
    $\phi$ is $\phi_1 \lor \phi_2$:  return NNF ($\phi_1$) $\lor$ NNF ($\phi_2$)
    $\phi$ is $\neg(\phi_1 \land \phi_2)$:  return NNF ($\neg\phi_1$) $\lor$ NNF ($\neg\phi_2$)
    $\phi$ is $\neg(\phi_1 \lor \phi_2)$:  return NNF ($\neg\phi_1$) $\land$ 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 CNF (NNF (IMPL_FREE (ϕ))). In the exercises, you are asked to show that

- all four algorithms terminate on input meeting their preconditions,
- the result of CNF (NNF (IMPL_FREE (ϕ))) 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 CNF (NNF (IMPL_FREE ($\neg p \land q \to p \land (r \to q)$))). 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

$\texttt{IMPL\_FREE}\ (\phi)$:

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

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

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

The formula $(p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q)$ 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 $p \vee \neg q$. 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 counter-productive.

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

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

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

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

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:

$$\begin{array}{rcl} P & ::= & \bot \mid \top \mid p \\ A & ::= & P \mid P \wedge A \\ C & ::= & A \rightarrow P \\ H & ::= & C \mid C \wedge H\,. \end{array} \tag{1.7}$$

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

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

$$(p \wedge q \wedge s \rightarrow \bot) \wedge (q \wedge r \rightarrow p) \wedge (\top \rightarrow s)$$

$$(p_2 \wedge p_3 \wedge p_5 \rightarrow p_{13}) \wedge (\top \rightarrow p_5) \wedge (p_5 \wedge p_{11} \rightarrow \bot).$$

Examples of formulas which are *not* Horn formulas are

$$(p \land q \land s \to \neg p) \land (q \land r \to p) \land (p \land s \to s)$$

$$(p \land q \land s \to \bot) \land (\neg q \land r \to p) \land (\top \to s)$$

$$(p_2 \land p_3 \land p_5 \to p_{13} \land p_{27}) \land (\top \to p_5) \land (p_5 \land p_{11} \to \bot)$$

$$(p_2 \land p_3 \land p_5 \to p_{13} \land p_{27}) \land (\top \to p_5) \land (p_5 \land p_{11} \lor \bot).$$

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, $\neg q \land r$, 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, $p_{13} \land p_{27}$, 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 $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ 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 $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ 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 $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ 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 $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ 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 $P_1 \land P_2 \land \cdots \land P_{k_i}$ have to be true in v as well. The conjunct $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ 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 $\square$

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:

$$\text{`All marked subformulas evaluate to their mark value} \\ \text{for all valuations in which } \phi \text{ evaluates to T.'} \tag{1.9}$$

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

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

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

$$\begin{array}{ll} T(p) = p & T(\neg\phi) = \neg T(\phi) \\ T(\phi_1 \land \phi_2) = T(\phi_1) \land T(\phi_2) & T(\phi_1 \lor \phi_2) = \neg(\neg T(\phi_1) \land \neg T(\phi_2)) \\ T(\phi_1 \to \phi_2) = \neg(T(\phi_1) \land \neg T(\phi_2)) & \end{array}$$

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 $p \land \neg(q \lor \neg p)$ we compute $T(\phi) = p \land \neg\neg(\neg q \land \neg\neg p)$. The parse tree and DAG of $T(\phi)$ are depicted in Figure 1.12.

Any valuation that makes $p \land \neg\neg(\neg q \land \neg\neg p)$ true has to assign `T` to the topmost $\land$-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 ($\neg$, $\land$ 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 $\land_{\mathrm{te}}$ propagates a `T` constraint on a $\land$-node to its two sub-nodes; dually, $\land_{\mathrm{ti}}$ forces a `T` mark on a $\land$-node if both its children have that mark. The laws $\land_{\mathrm{fl}}$ and $\land_{\mathrm{fr}}$ force a `F` constraint on a $\land$-node if any of its sub-nodes has a `F` value. The laws $\land_{\mathrm{fll}}$ and $\land_{\mathrm{frr}}$ are more complex: if an $\land$-node has a `F` constraint and one of its sub-nodes has a `T` constraint, then the *other* sub-node obtains a `F`-constraint.

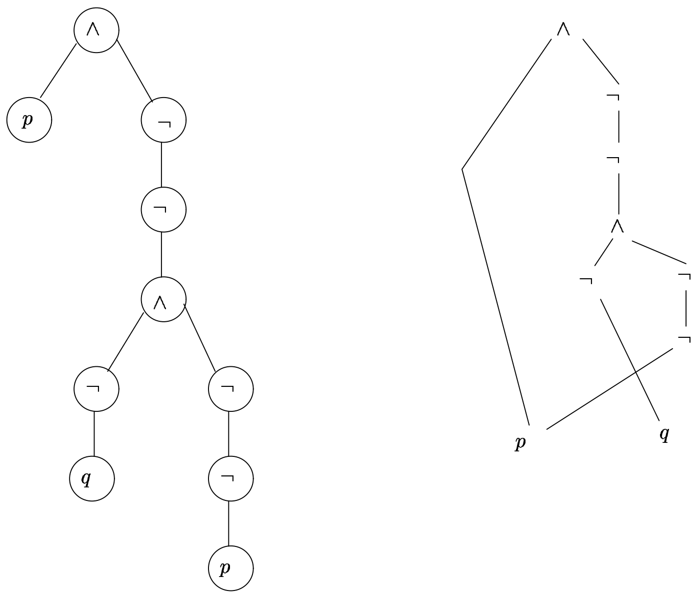


Fig. 1.12. Parse tree (left) and directed acyclic graph (right) of the formula from Example 1.48. The p-node is shared on the right.

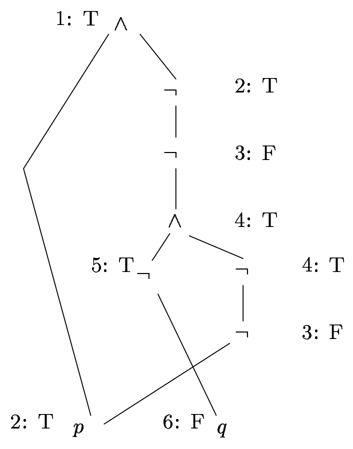


Fig. 1.13. A witness to the satisfiability of the formula represented by this DAG.

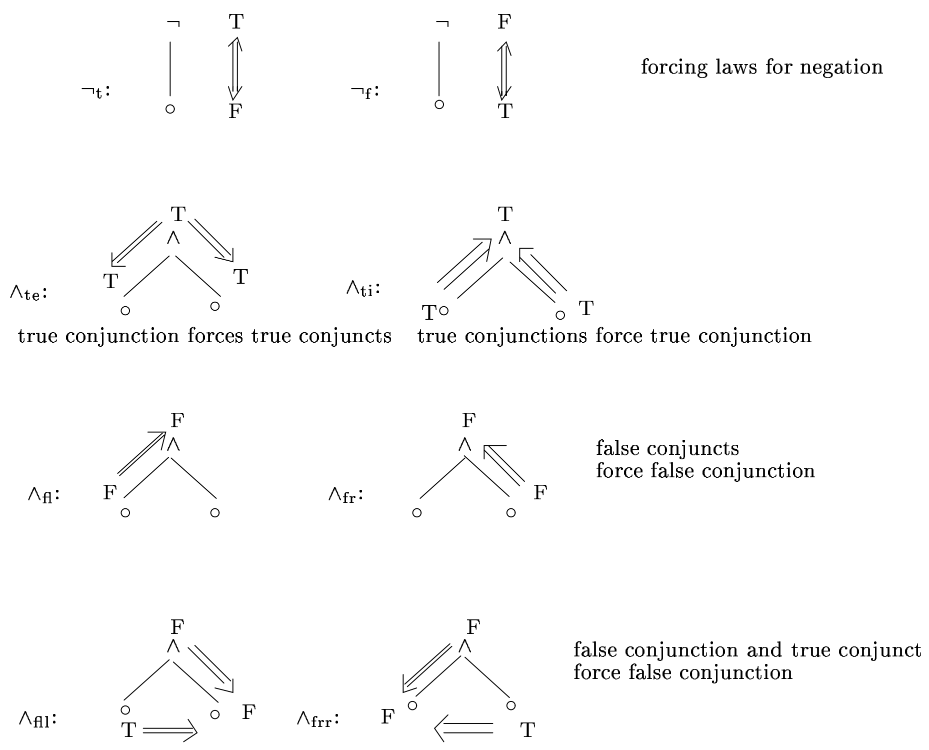

$\neg_t$:
$\neg_f$:
forcing laws for negation
$\wedge_{te}$:
$\wedge_{ti}$:
true conjunction forces true conjuncts
true conjunctions force true conjunction
$\wedge_{fl}$:
$\wedge_{fr}$:
false conjuncts force false conjunction
$\wedge_{fll}$:
$\wedge_{frr}$:
false conjunction and true conjunct force false conjunction

Fig. 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 $\wedge_{fll}$, $\wedge_{frr}$ and $\wedge_{ti}$ require that the source constraints of both $\Longrightarrow$ are present.

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 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.

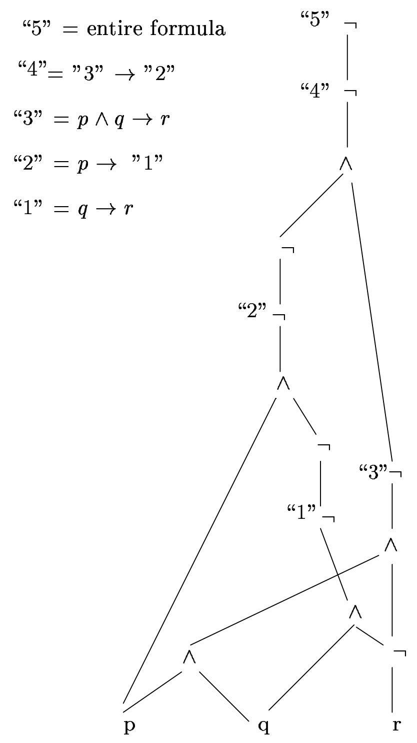


Fig. 1.15. The DAG for the translation of $\neg((p \wedge q \to r) \to p \to q \to r)$. Labels '1' etc indicate which nodes represent what subformulas.

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 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

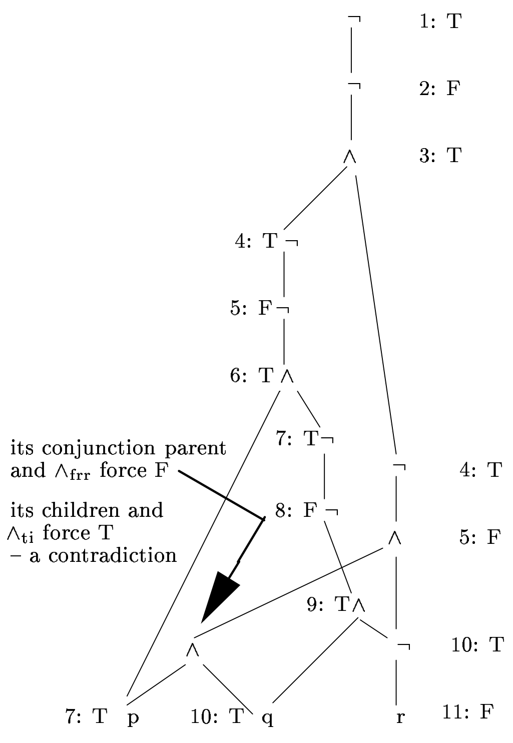

1: T
2: F
3: T
4: T
5: F
6: T
7: T
8: F
9: T
its conjunction parent and ∧frr force F
its children and ∧ti force T – a contradiction
4: T
5: F
10: T
11: F
7: T p
10: T q
r

Fig. 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.

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 $\neg(\phi_1 \land \phi_2)$.

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

$$((p \lor (q \lor r)) \land ((p \lor \neg q) \land ((q \lor \neg r) \land ((r \lor \neg p) \land (\neg p \lor (\neg q \lor \neg r)))))) \tag{1.11}$$

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

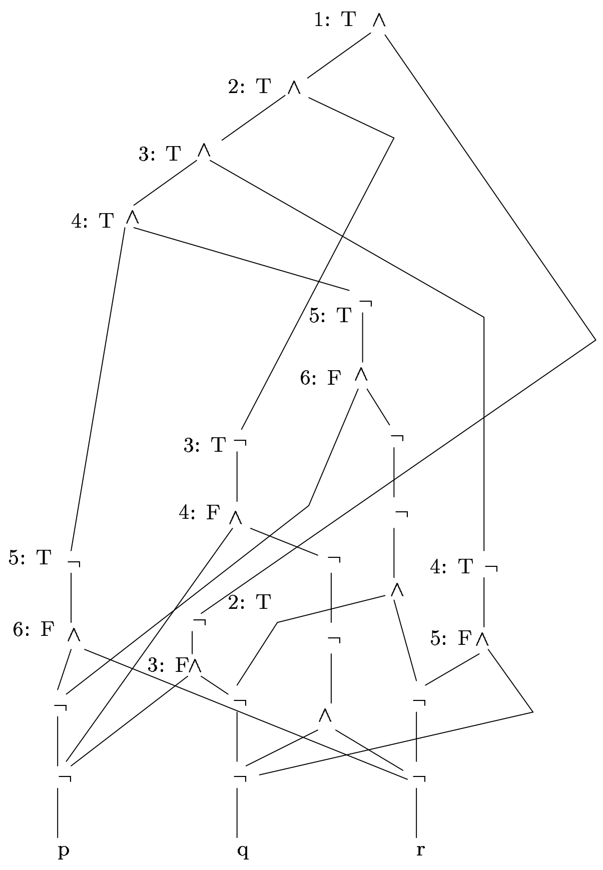


Fig. 1.17. The DAG for the translation of the formula in (1.11). It has a $\wedge$-spine of length 4 as it is a conjunction of five clauses. Its linear analysis gets stuck: all forced constraints are consistent with each other but several nodes, including all atoms, are unconstrained.

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 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.

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 $\neg$-node and explore the consequences of setting that $\neg$-node's mark to `T`; Figure 1.18 shows the result of that analysis. Dually, Figure 1.19 tests the consequences of setting that $\neg$-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.

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.

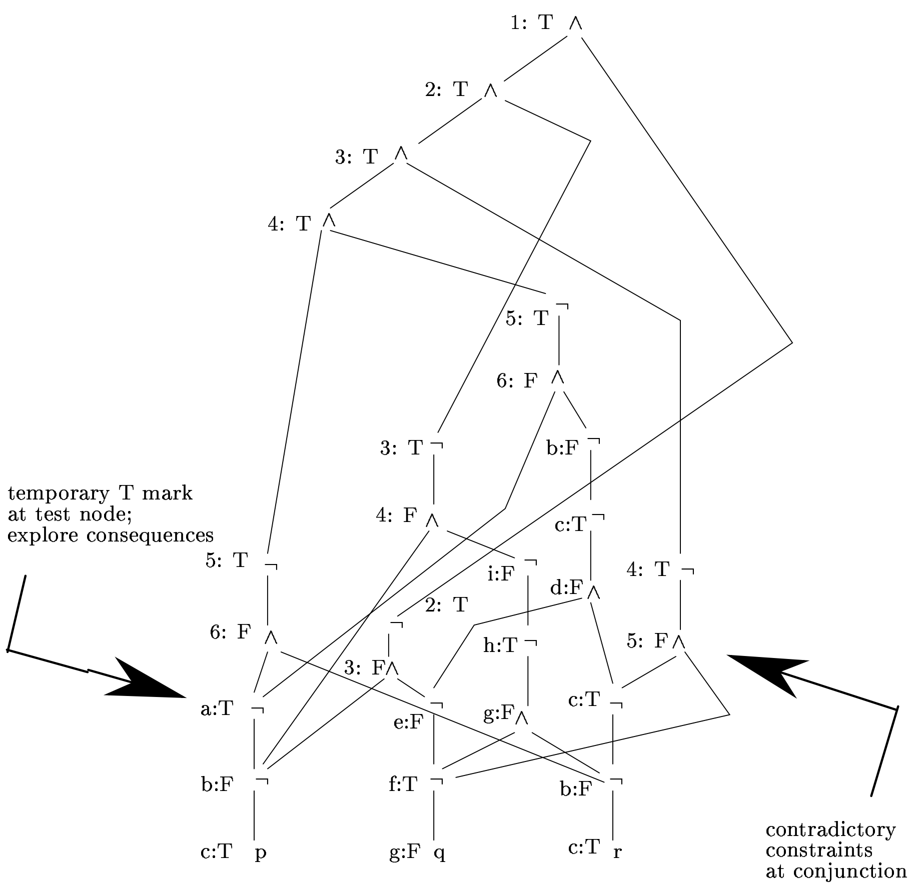


Fig. 1.18. Marking an unmarked node with T and exploring what new constraints would follow from this. The analysis shows that this test marking causes contradictory constraints. We use lowercase letters 'a:' etc to denote temporary marks.

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.

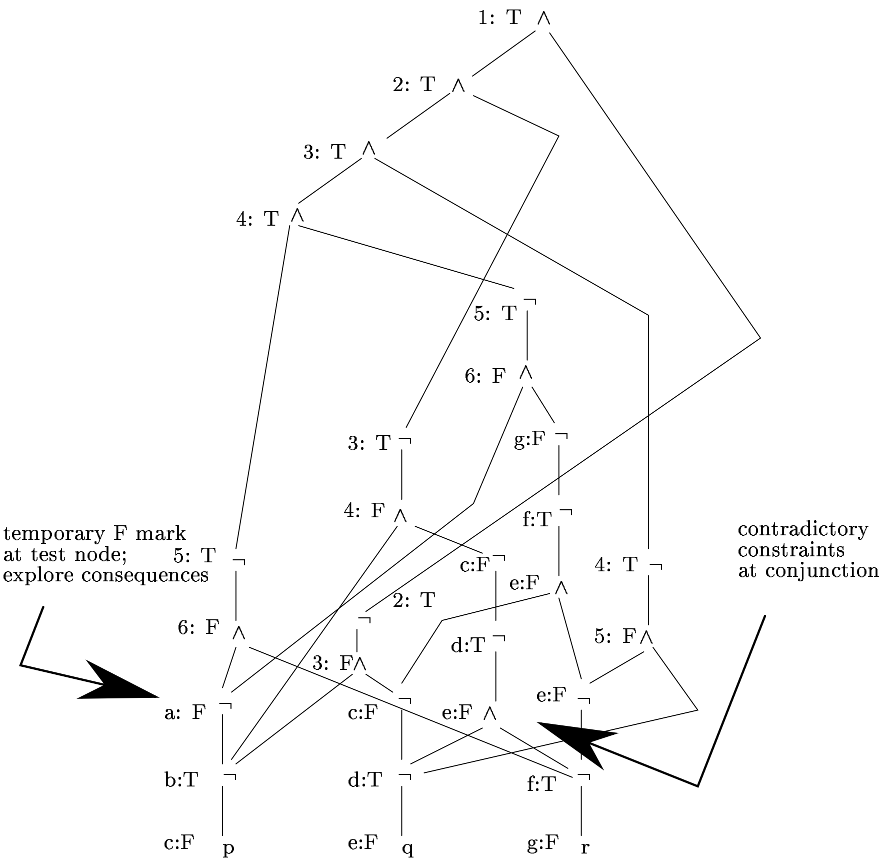


Fig. 1.19. Marking the same unmarked node with F and exploring what new constraints would follow from this. The analysis shows that this test marking also causes contradictory constraints.

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.

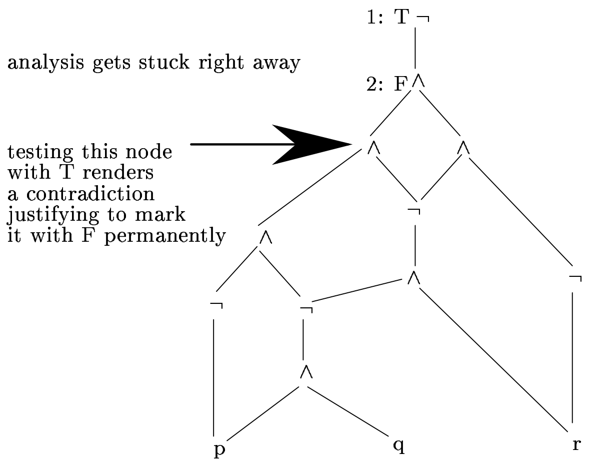

1: T ¬
analysis gets stuck right away
2: F ∧
testing this node with T renders a contradiction justifying to mark it with F permanently
p
q
r

Fig. 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.

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 $p \wedge q \to r$, change it to $(p \wedge q) \to r$ since $\wedge$ binds more tightly than $\to$.
   - * (a) $\neg p \wedge q \to r$
   - (b) $(p \to q) \wedge \neg(r \vee p \to q)$
   - * (c) $(p \to q) \to (r \to s \vee t)$
   - (d) $p \vee (\neg q \to p \wedge r)$
   - * (e) $p \vee q \to \neg p \wedge r$
   - (f) $p \vee p \to \neg q$
   - * (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) $p \to (p \to q), p \vdash q$
   - * (e) $q \to (p \to r), \neg r, q \vdash \neg p$
   - * (f) $\vdash (p \wedge q) \to p$
   - (g) $p \vdash q \to (p \wedge q)$
   - * (h) $p \vdash (p \to q) \to q$
   - * (i) $(p \to r) \wedge (q \to r) \vdash p \wedge q \to r$
   - * (j) $q \to r \vdash (p \to q) \to (p \to r)$
   - (k) $p \to (q \to r), p \to q \vdash p \to r$
   - * (l) $p \to q, r \to s \vdash p \vee r \to q \vee s$
   - (m) $p \vee q \vdash r \to (p \vee q) \wedge r$
   - * (n) $(p \vee (q \to p)) \wedge q \vdash p$
   - * (o) $p \to q,\ r \to s \vdash p \wedge r \to q \wedge s$
   - (p) $p \to q \vdash ((p \wedge q) \to p) \wedge (p \to (p \wedge q))$
   - (q) $\vdash q \to (p \to (p \to (q \to p)))$
   - * (r) $p \to q \wedge r \vdash (p \to q) \wedge (p \to r)$
   - (s) $(p \to q) \wedge (p \to r) \vdash p \to q \wedge r$
   - (t) $\vdash (p \to q) \to ((r \to s) \to (p \wedge r \to q \wedge s))$; here you might be able to 'recycle' and augment a proof from a previous exercise.
   - (u) $p \to q \vdash \neg q \to \neg p$
   - * (v) $p \vee (p \wedge q) \vdash p$

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

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

* (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) $\neg p \to \neg q \vdash q \to p$

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

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

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

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

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

3. Prove the validity of the sequents below:

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

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

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

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

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

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

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

(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 21

(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 $\phi_1, \phi_2, \ldots, \phi_n$ $(n \geq 0)$. 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 $\to$i. The new approach expresses this as an inference rule between sequents:

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

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

* 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:

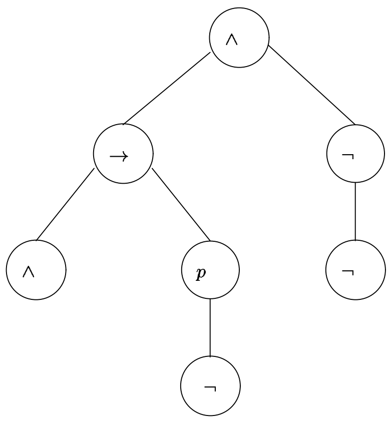


Fig. 1.21. A tree that represents an ill-formed formula.

(a) the tree in Figure 1.10 on page 45
(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) $p \wedge \neg(p \vee \neg q) \to (r \to s)$
(b) $p \wedge \neg(p \vee q \wedge s) \to (r \to s)$
(c) $p \wedge \neg(p \vee \wedge s) \to (r \to s)$.

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 $p \to q$. (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) $((p \to q) \to p) \to p$
(b) represented by the parse tree in Figure 1.3 on page 35

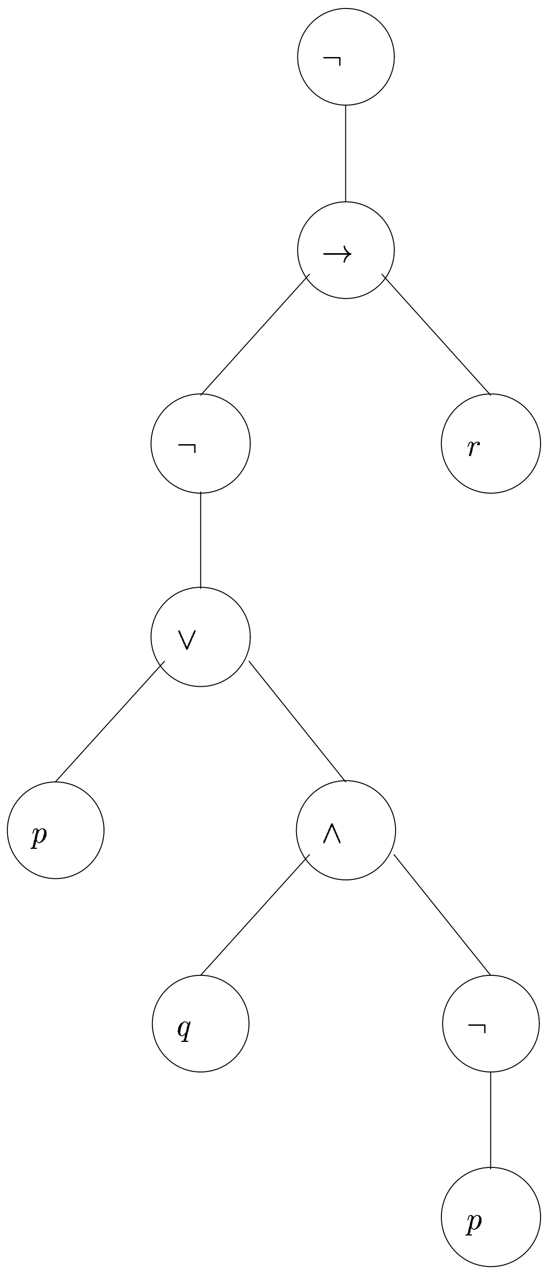


Fig. 1.22. A parse tree of a negated implication.

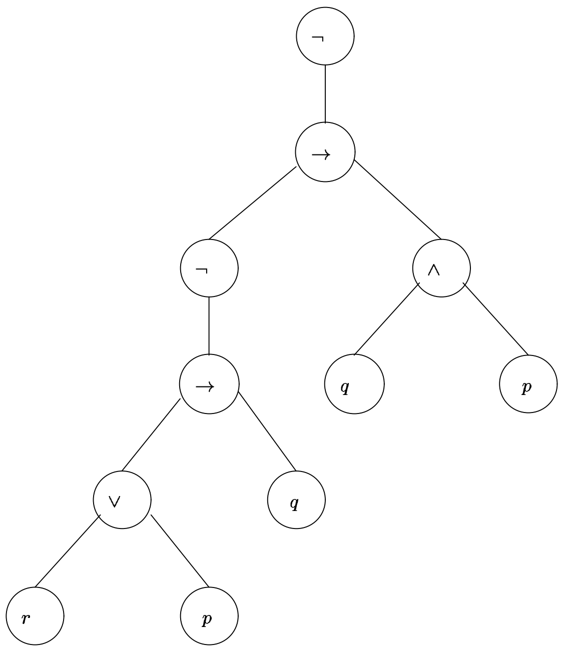


Fig. 1.23. Another parse tree of a negated implication.

* (c) $p \lor (\neg(q \land (r \to q)))$
(d) $(p \land q) \to (p \lor q)$
(e) $((p \to \neg q) \to \neg p) \to q$
(f) $(p \to q) \lor (p \to \neg q)$
(g) $((p \to q) \to p) \to p$
(h) $((p \lor q) \to r) \to ((p \to r) \lor (q \to r))$
(i) $(p \to q) \to (\neg p \to \neg q)$.

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 45 and the valuation in which q and r evaluate to `T` and p to `F`;
   (b) Figure 1.4 on page 37 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 45 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 39)? 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, $\circ \in \{\to, \vee, \wedge\}$ 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 47,

(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 T and the truth value of the formula to the right of $\vdash$ is F.

    (a) $\neg p \vee (q \to p) \vdash \neg p \wedge q$
    (b) $\neg r \to (p \vee q), r \wedge \neg q \vdash r \to q$
    * (c) $p \to (q \to r) \vdash p \to (r \to q)$
    (d) $\neg p, p \vee q \vdash \neg q$
    (e) $p \to (\neg q \vee r), \neg r \vdash \neg q \to \neg p$.

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) $\neg p \to \neg q \vdash \neg q \to \neg p$
    (c) $p \to q \vdash p \vee q$
    (d) $p \to (q \vee r) \vdash (p \to q) \wedge (p \to r)$.

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 $((\phi_1 \wedge (\phi_2 \wedge (\cdots \wedge \phi_n)\ldots) \to \psi) \to (\phi_1 \to (\phi_2 \to (\ldots(\phi_n \to \psi)\ldots))))$.
16. Prove the validity of the following sequents needed to secure the completeness result for propositional logic:

    (a) $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \to \phi_2)$
    (b) $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \to \phi_2$
    (c) $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$
    (d) $\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$
    (e) $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$
    (f) $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$
    (g) $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$
    (h) $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \vee \phi_2)$
    (i) $\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$
    (j) $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$
    (k) $\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \vee \phi_2$.

17. Does $\vDash \phi$ hold for the ϕ below? Please justify your answer.

    (a) $(p \to q) \vee (q \to r)$
    * (b) $((q \to (p \vee (q \to p))) \vee \neg(p \to q)) \to p$.

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) $\land$ and $\lor$ are idempotent:
     - (i) $\phi \land \phi \equiv \phi$
     - (ii) $\phi \lor \phi \equiv \phi$
   - (b) $\land$ and $\lor$ are commutative:
     - (i) $\phi \land \psi \equiv \psi \land \phi$
     - (ii) $\phi \lor \psi \equiv \psi \lor \phi$
   - (c) $\land$ and $\lor$ 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 33.

* 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 57.

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 68 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) $(p \wedge q \wedge w \to \bot) \wedge (t \to \bot) \wedge (r \to p) \wedge (\top \to r) \wedge (\top \to q) \wedge (r \wedge u \to w) \wedge (u \to s) \wedge (\top \to u)$
(c) $(p \wedge q \wedge s \to p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$
(d) $(p \wedge q \wedge s \to \bot) \wedge (q \wedge r \to p) \wedge (\top \to s)$
(e) $(p_5 \to p_{11}) \wedge (p_2 \wedge p_3 \wedge p_5 \to p_{13}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \to \bot)$
(f) $(\top \to q) \wedge (\top \to s) \wedge (w \to \bot) \wedge (p \wedge q \wedge s \to \bot) \wedge (v \to s) \wedge (\top \to r) \wedge (r \to p)$
* (g) $(\top \to q) \wedge (\top \to s) \wedge (w \to \bot) \wedge (p \wedge q \wedge s \to v) \wedge (v \to s) \wedge (\top \to r) \wedge (r \to p)$.

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 67 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 34,
   (a) $T(\phi)$ can be generated by (1.10) on page 71,
   (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 73) are sound: if all current marks satisfy the invariant (1.9) from page 70, then this invariant still holds if the derived constraint of that rule becomes an additional mark.

3. In Figure 1.16 on page 75 we detected a contradiction which secured the validity of the sequent $p \wedge q \to r \vdash p \to q \to r$. Use the same method with the linear SAT solver to show that the sequent $\vdash (p \to q) \vee (r \to p)$ 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 $p \vee q, p \to r \vdash r$. 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 $p \vee q, p \to r \vdash r$ 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 80. 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 76) is indeed the one obtained for $T(\phi)$, where ϕ is the formula in (1.11) on page 75.

* 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 $\cdot$ 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].

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) focusses 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 $p, q, r, \ldots$, or $p_1, p_2, \ldots$). 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:

$$\begin{aligned}\phi \ ::= \ & \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \\ & \mid (\mathrm{X}\,\phi) \mid (\mathrm{F}\,\phi) \mid (\mathrm{G}\,\phi) \mid (\phi \mathbin{\mathrm{U}} \phi) \mid (\phi \mathbin{\mathrm{W}} \phi) \mid (\phi \mathbin{\mathrm{R}} \phi)\end{aligned} \qquad (3.1)$$

where p is any propositional atom from some set `Atoms`.

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, W and R 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:

- $(((\mathrm{F}\,p) \wedge (\mathrm{G}\,q)) \rightarrow (p \mathbin{\mathrm{W}} r))$
- $(\mathrm{F}\,(p \rightarrow (\mathrm{G}\,r)) \vee ((\neg q) \mathbin{\mathrm{U}} p))$, the parse tree of this formula is illustrated in Figure 3.1.
- $(p \mathbin{\mathrm{W}} (q \mathbin{\mathrm{W}} r))$
- $((\mathrm{G}\,(\mathrm{F}\,p)) \rightarrow (\mathrm{F}\,(q \vee s)))$.

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.

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 $\rightarrow$.

These binding priorities allow us to drop some brackets without introducing ambiguity. The examples above can be written

- $\mathrm{F}\,p \wedge \mathrm{G}\,q \rightarrow p \mathbin{\mathrm{W}} r$
- $\mathrm{F}\,(p \rightarrow \mathrm{G}\,r) \vee \neg q \mathbin{\mathrm{U}} p$
- $p \mathbin{\mathrm{W}} (q \mathbin{\mathrm{W}} r)$
- $\mathrm{G}\,\mathrm{F}\,p \rightarrow \mathrm{F}\,(q \vee s)$.

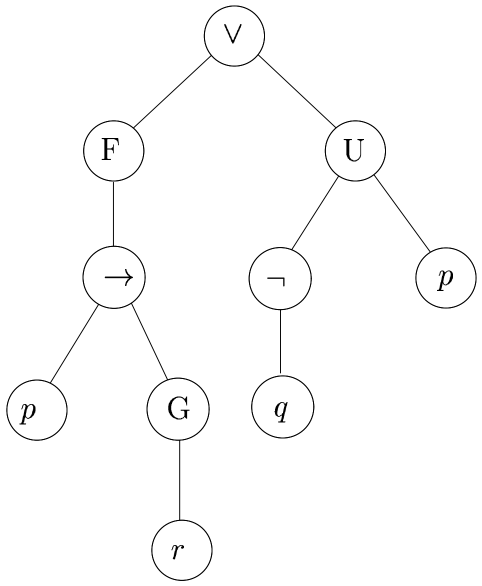


Fig. 3.1. The parse tree of $(\mathrm{F}\,(p \rightarrow \mathrm{G}\,r) \vee (\neg q\ \mathrm{U}\ p))$.

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 $\mathrm{F}\,p \rightarrow \mathrm{G}\,r \vee \neg q\ \mathrm{U}\ p$, 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:

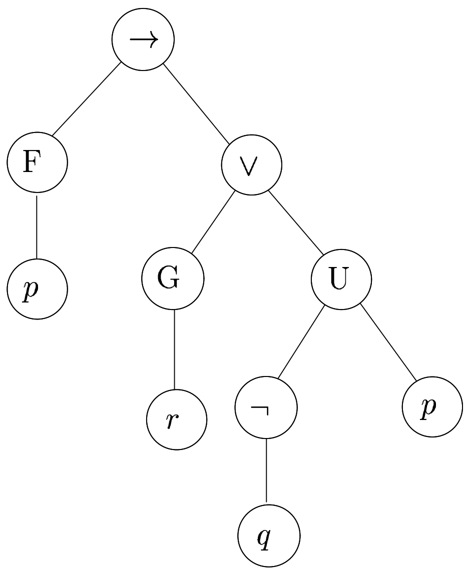


Fig. 3.2. The parse tree of $\mathrm{F}\, p \to \mathrm{G}\, r \lor \neg q\ \mathrm{U}\ p$, assuming binding priorities of Convention 3.2.

Definition 3.4 A transition system $\mathcal{M} = (S, \to, L)$ is a set of states S endowed with a transition relation $\to$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \to s'$, and a labelling function $L\colon S \to \mathcal{P}(\mathtt{Atoms})$.

Transition systems are also simply called *models* in this chapter. So a model has a collection of states S, a relation $\to$, 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 $s_0 \to s_1$, $s_0 \to s_2$, $s_1 \to s_0$, $s_1 \to s_2$ and $s_2 \to s_2$; and

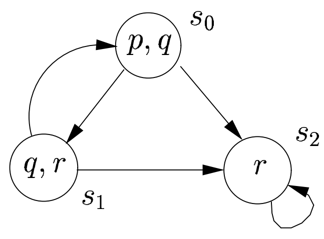


Fig. 3.3. A concise representation of a transition system $\mathcal{M} = (S, \rightarrow, L)$ as a directed graph. We label state s with l iff $l \in L(s)$.

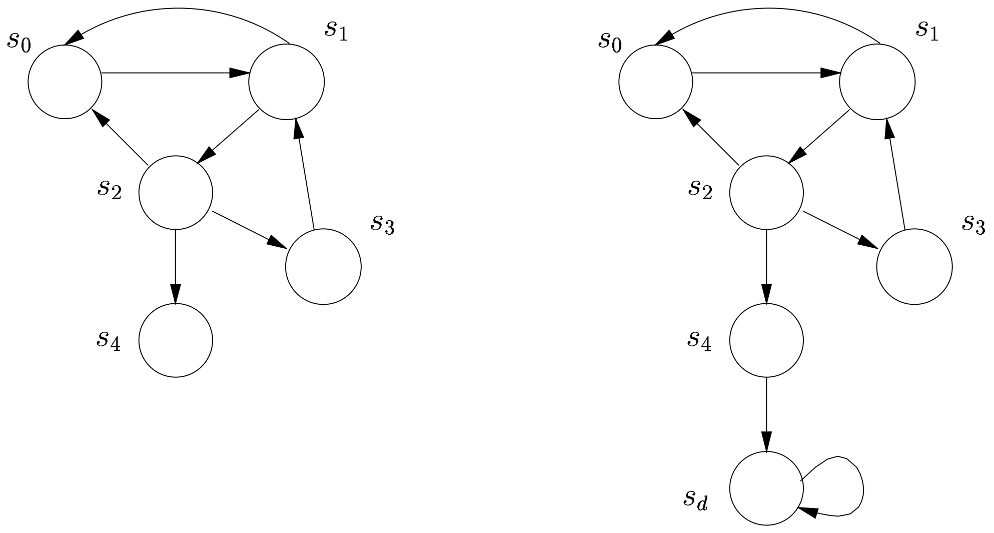


Fig. 3.4. On the left, we have a system with a state s_4 that does not have any further transitions. On the right, we expand that system with a 'deadlock' state s_d such that no state can deadlock; of course, it is then our understanding that reaching the 'deadlock' state s_d corresponds to deadlock in the original system.

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 transitions $s \rightarrow s_d$ for each s which was a deadlock in the old system, as well as $s_d \rightarrow s_d$. See Figure 3.4 for such an example.

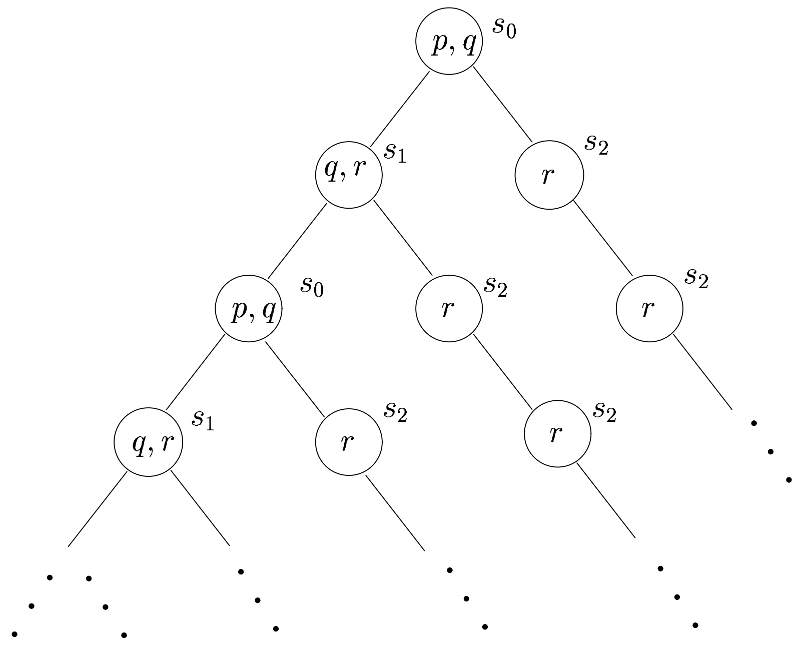


Fig. 3.5. Unwinding the system of Figure 3.3 as an infinite tree of all computation paths beginning in a particular state.

Definition 3.5 A path in a model $\mathcal{M} = (S, \rightarrow, L)$ is an infinite sequence of states $s_1, s_2, s_3, \dots$ in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$. We write the path as $s_1 \rightarrow s_2 \rightarrow \dots$.

Consider the path $\pi = s_1 \rightarrow s_2 \rightarrow \dots$. 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 $s_3 \rightarrow s_4 \rightarrow \dots$.

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 $\pi = s_1 \rightarrow \dots$ be a path in $\mathcal{M}$. Whether π satisfies an LTL formula is defined by the satisfaction relation $\models$ as follows:

1. $\pi \models \top$
2. $\pi \not\models \bot$

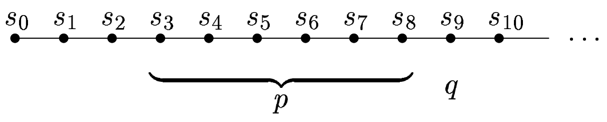

s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10}
...
p
q

Fig. 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 U q along the path shown.

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. $\pi \vDash \phi_1 \to \phi_2$ 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 $i \geq 1$, $\pi^i \vDash \phi$
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 $j = 1, \ldots, i-1$ 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 $j = 1, \ldots, i-1$ 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 $j = 1, \ldots, i$ 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 G $(p \to X\,q)$, 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 ϕ_1 U ϕ_2 holds on a path if it is the case that ϕ_1 holds continuously *until* ϕ_2 holds. Moreover, ϕ_1 U ϕ_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 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 ϕ W ψ 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 elevator at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
  $\mathrm{G}\,(\mathsf{floor{=}2} \wedge \mathsf{direction{=}up} \wedge \mathsf{ButtonPressed5} \to (\mathsf{direction{=}up}\ \mathrm{U}\ \mathsf{floor{=}5}))$
  Here, our atomic descriptions are boolean expressions built from system variables, e.g. floor=2.

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 elevator *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:

$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi \qquad\qquad \neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi \ .$$

(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:

$$\neg \mathrm{G}\,\phi \equiv \mathrm{F}\,\neg\phi \qquad\qquad \neg \mathrm{F}\,\phi \equiv \mathrm{G}\,\neg\phi \qquad\qquad \neg \mathrm{X}\,\phi \equiv \mathrm{X}\,\neg\phi \ .$$

Also U and R are duals of each other:

$$\neg(\phi \,\mathrm{U}\, \psi) \equiv \neg\phi \,\mathrm{R}\, \neg\psi \qquad\qquad \neg(\phi \,\mathrm{R}\, \psi) \equiv \neg\phi \,\mathrm{U}\, \neg\psi \ .$$

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:

$$\mathrm{F}\,\phi \equiv \top\ \mathrm{U}\ \phi \qquad\qquad \mathrm{G}\,\phi \equiv \bot\ \mathrm{R}\,\phi\ .$$

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 $\mathrm{G}\,\phi \equiv \bot\ \mathrm{R}\ \phi$, 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:

$$\phi\ \mathrm{U}\ \psi \equiv \phi\ \mathrm{W}\ \psi \wedge \mathrm{F}\,\psi\,. \tag{3.2}$$

To prove equivalence (3.2), suppose first that a path satisfies ϕ U ψ. 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 ϕ W ψ, 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 $\phi \mathrel{\mathrm{R}} \psi \equiv \neg(\neg\phi \mathrel{\mathrm{U}} \neg\psi)$ and equivalence (3.4) followed by the duality, respectively.
  - U and W may be defined from R, by the duality $\phi \mathrel{\mathrm{U}} \psi \equiv \neg(\neg\phi \mathrel{\mathrm{R}} \neg\psi)$ and equivalence (3.4), respectively.
  - R and U may be defined from W, by equivalence (3.5) and the duality $\phi \mathrel{\mathrm{U}} \psi \equiv \neg(\neg\phi \mathrel{\mathrm{R}} \neg\psi)$ 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 proposition 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 $\phi \mathrel{\mathrm{U}} \psi \quad\equiv\quad \neg(\neg\psi \mathrel{\mathrm{U}} (\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi$ holds for all LTL formulas ϕ and ψ.

Proof: Take any path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ in any model.

First, suppose $s_0 \vDash \phi \mathrel{\mathrm{U}} \psi$ holds. Let n be the smallest number such that $s_n \vDash \psi$; such a number has to exist since $s_0 \vDash \phi \mathrel{\mathrm{U}} \psi$; then, for each $k < n$, $s_k \vDash \phi$. We immediately have $s_0 \vDash \mathrm{F}\,\psi$, so it remains to show $s_0 \vDash \neg(\neg\psi \mathrel{\mathrm{U}} (\neg\phi \wedge \neg\psi))$, which, if we expand, means:

(∗) for each $i > 0$, if $s_i \vDash \neg\phi \wedge \neg\psi$, then there is some $j < i$ with $s_j \vDash \psi$.

Take any $i > 0$ with $s_i \vDash \neg\phi \wedge \neg\psi$; $i > n$, so we can take $j \stackrel{\text{def}}{=} n$ and have $s_j \vDash \psi$.

Conversely, suppose $s_0 \vDash \neg(\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)) \wedge \text{F}\,\psi$ holds; we prove $s_0 \vDash \phi \text{ U } \psi$. Since $s_0 \vDash \text{F}\,\psi$, 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. $\square$

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.

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.

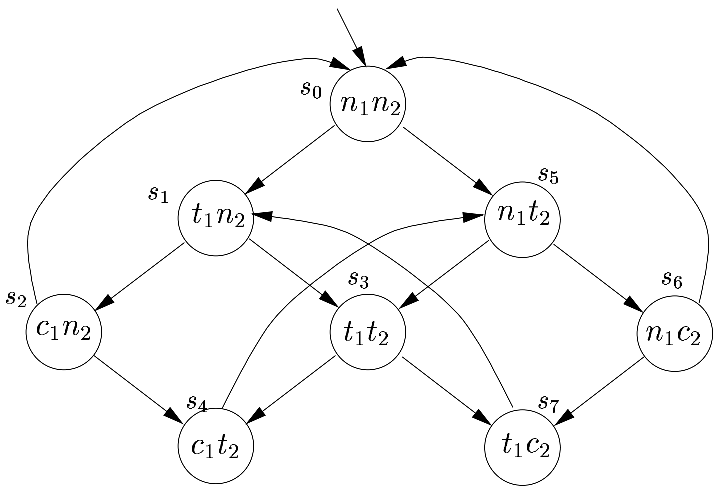


Fig. 3.7. A first-attempt model for mutual exclusion.

3.3.1.1 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 path starting at the initial sate 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 224).

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: $\mathrm{G}\,(c_1 \to c_1 \;\mathrm{W}\; (\neg c_1 \land \neg c_1 \;\mathrm{W}\; c_2))$. 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.

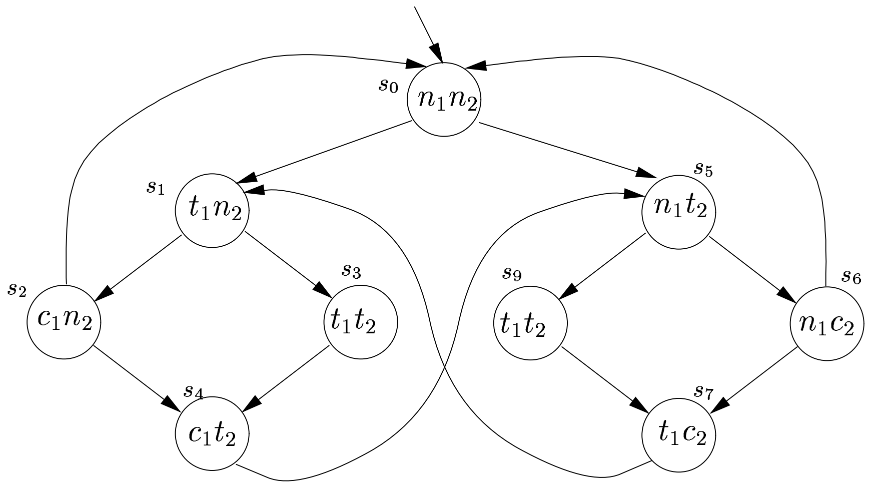


Fig. 3.8. A second-attempt model for mutual exclusion. There are now two states representing t_1t_2, namely s_3 and s_9.

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.

3.3.1.2 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_1t_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).

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};
```

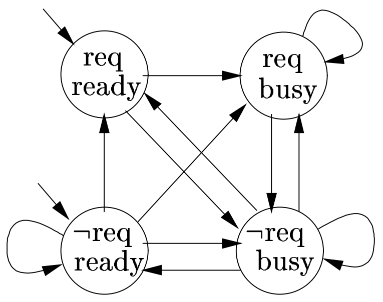


Fig. 3.9. The model corresponding to the SMV program in the text.

```
                    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 '¬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

¬, respectively, since they are available on standard keyboards. We may easily verify that the specification of our module `main` holds of the model in Figure 3.9.

3.3.2.1 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.

3.3.2.2 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 264. 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 3.3.1.2).

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 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

```
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)
```

Fig. 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.

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 (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 ferry-man

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 behavioral 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 behavior (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, etc. 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` $\neq$ `cabbage` then that set is $\{\mathtt{g}, \mathtt{w}, \mathtt{0}\}$. 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.

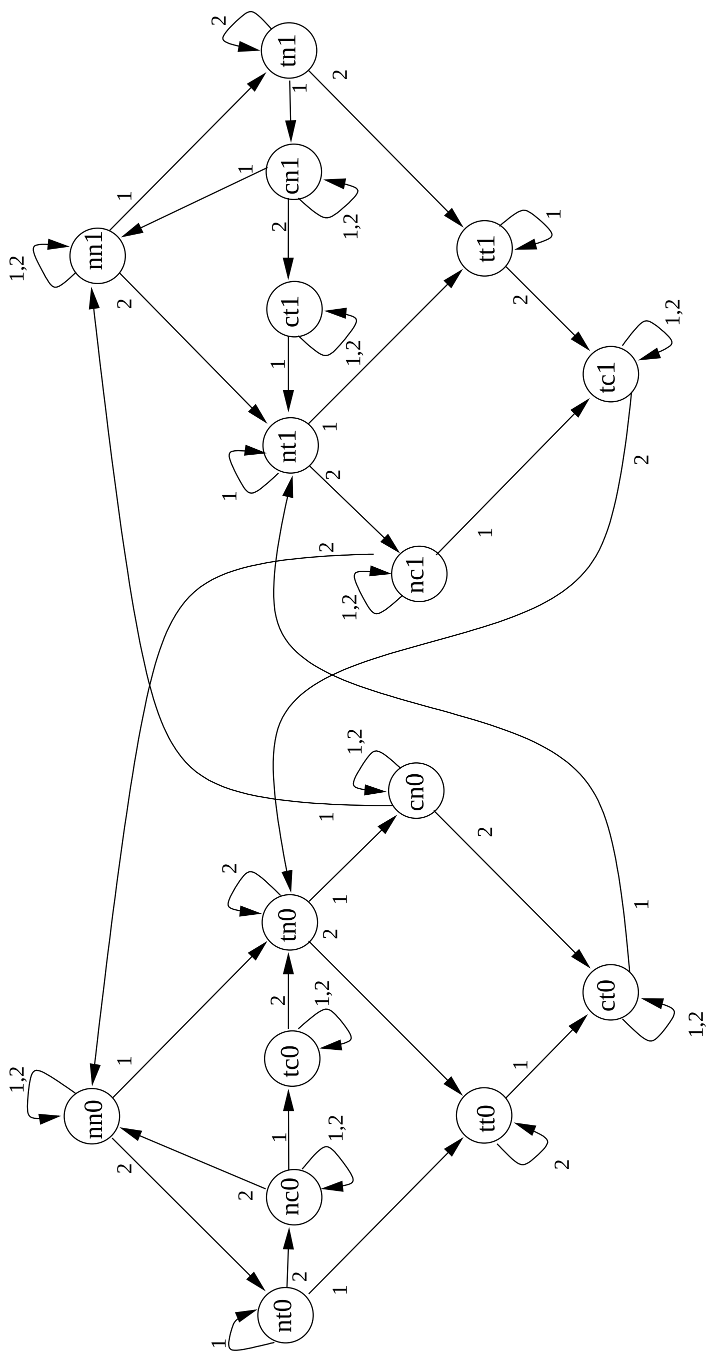


Fig. 3.11. The transition system corresponding to the SMV code in Figure 3.10. The labels on the transitions denote the process which makes the move. The label 1, 2 means that either process could make that move.

```
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))
```

Fig. 3.12. NuSMV code for the ferry-man planning problem.

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 pesimistic: sometimes NuSMV complains of circularity even in situations when it could be resolved. The original CMU-SMV is more liberal in this respect.)

3.3.5.1 Running NuSMV

We seek a path satisfying ϕ U ψ, 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 $\neg(\phi \text{ U } \psi)$, i.e. no path satisfies ϕ U ψ. We hope this is not the case, and NuSMV will give us an example path which does satisfy ϕ U ψ. 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 ϕ U ψ, 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 ϕ U ψ (with ψ occurring at state 8). What happens along the path after that has no bearing on ϕ U ψ.

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 7 ferryman.smv` shows that the LTL formula holds in that model, meaning that no solution of length ≤ 7 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 ($\texttt{goat} = \texttt{cabbage} \vee \texttt{goat} =$

```
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
```

Fig. 3.13. A solution path to the ferryman puzzle. It is unnecessarily long. Using bounded model checking will refine it into an optimal solution.

wolf) $\rightarrow$ goat $=$ ferryman and ψ equals cabbage $\wedge$ goat $\wedge$ wolf $\wedge$ ferryman, we now seek a path satisfying $(\phi \text{ U } \psi) \wedge \text{G}\,($goat $\rightarrow \text{G}\,$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 ABP (alternating bit protocol) 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 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

```
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
```

Fig. 3.14. The ABP sender in SMV.

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 non-deterministic assignment: the `input` may be transmitted to the `output`, but it need not (in which case output retains its old value). However, the second fairness constraint ensures that the channel doesn't continually lose the same message: eventually, a bit will get through (so if `input` is 1, then eventually `output` will be 1 too).

The two-bit channel `two-bit-chan`, used to send messages, is similar. 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).

A fairness constraint models the fact that, although channels can lose messages, even infinitely often, we assume that they infinitely often transmit

```
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
```

Fig. 3.15. The ABP receiver in SMV.

the message correctly. (If this were not the case, then we could find an uninteresting violation of the liveness constraint, for example a path along which all messages from a certain time onwards get lost.)

Finally, we tie it all together with the module `main` (Figure 3.17). Its role is to connect together the components of the system, which it does by instantiating the four processes. It also specifies the initial values. 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 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 received by the receiver: `G (S.st=sent & S.message1=1 -> msg_chan.output1=1)`.

```
MODULE one-bit-chan(input)
VAR
   output : boolean;
ASSIGN
   next(output) := {input,output};
FAIRNESS running
FAIRNESS (input=0 -> AF output=0) & (input=1 -> AF output=1)

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 !forget
```

Fig. 3.16. The two modules for the two ABP channels in SMV.

```
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)
```

Fig. 3.17. The `main` ABP module.

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 $\mathrm{AG}\,(p \to \mathrm{E}[p \;\mathrm{U}\; q])$.
- 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:

$$\begin{aligned}\phi ::= \quad & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi \mid \\ & \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{A}[\phi \mathrel{\mathrm{U}} \phi] \mid \mathrm{E}[\phi \mathrel{\mathrm{U}} \phi]\end{aligned}$$

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 $\to$, 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:

- $\mathrm{AG}\,(q \to \mathrm{EG}\,r)$, note that this is not the same as $\mathrm{AG}\,q \to \mathrm{EG}\,r$, for according to Convention 3.13, the latter formula means $(\mathrm{AG}\,q) \to (\mathrm{EG}\,r)$
- $\mathrm{EF}\,\mathrm{E}[r \mathrel{\mathrm{U}} q]$
- $\mathrm{A}[p \mathrel{\mathrm{U}} \mathrm{EF}\,r]$

- $\mathrm{EF}\,\mathrm{EG}\,p \to \mathrm{AF}\,r$, again, note that this binds as $(\mathrm{EF}\,\mathrm{EG}\,p) \to \mathrm{AF}\,r$, not $\mathrm{EF}\,(\mathrm{EG}\,p \to \mathrm{AF}\,r)$ or $\mathrm{EF}\,\mathrm{EG}\,(p \to \mathrm{AF}\,r)$
- $\mathrm{A}[p_1\ \mathrm{U}\ \mathrm{A}[p_2\ \mathrm{U}\ p_3]]$
- $\mathrm{E}[\mathrm{A}[p_1\ \mathrm{U}\ p_2]\ \mathrm{U}\ p_3]$
- $\mathrm{AG}\,(p \to \mathrm{A}[p\ \mathrm{U}\ (\neg p \wedge \mathrm{A}[\neg p\ \mathrm{U}\ q])])$.

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 $\mathrm{EF}\,\mathrm{A}[r\ \mathrm{U}\ q]$.

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 $\mathrm{A}[(r\ \mathrm{U}\ q) \wedge (p\ \mathrm{U}\ r)]$ 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 $\mathrm{A}[\phi\ \mathrm{U}\ \psi]$. Now, the ϕ and the ψ *may* contain $\wedge$, since they are arbitrary formulas; so $\mathrm{A}[(p \wedge q)\ \mathrm{U}\ (\neg r \to q)]$ *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 $\phi_1\ \mathrm{AU}\ \phi_2$, 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 $\mathrm{A}[\mathrm{AX}\,\neg p\ \mathrm{U}\ \mathrm{E}[\mathrm{EX}\,(p \wedge q)\ \mathrm{U}\ \neg p]]$ 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.

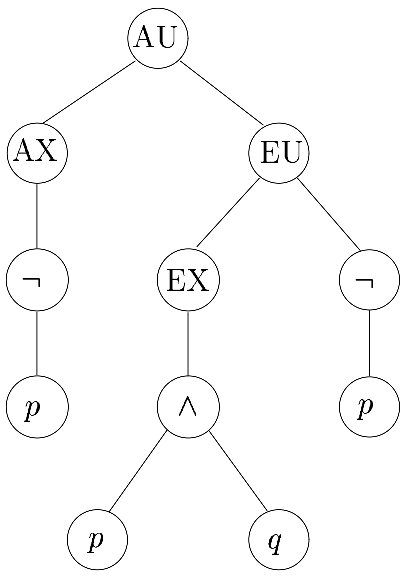


Fig. 3.18. The parse tree of a CTL formula without infix notation.

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 $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, 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 $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, 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.
11. $\mathcal{M}, s \vDash \mathrm{AF}\,\phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow \ldots$, 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 $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, 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. $\mathcal{M}, s \vDash \mathrm{A}[\phi_1 \mathrel{\mathrm{U}} \phi_2]$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, that path satisfies $\phi_1 \mathrel{\mathrm{U}} \phi_2$, 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. $\mathcal{M}, s \vDash \mathrm{E}[\phi_1 \mathrel{\mathrm{U}} \phi_2]$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$, where s_1 equals s, and that path satisfies $\phi_1 \mathrel{\mathrm{U}} \phi_2$ as specified in 13. Mnemonically: there Exists a computation path beginning in s such that ϕ_1 Until ϕ_2 holds on it.

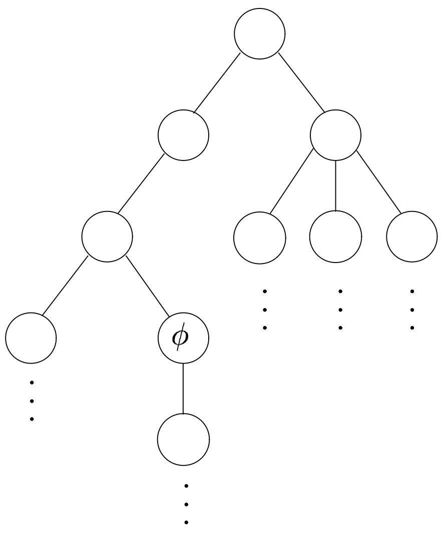


Fig. 3.19. A system whose starting state satisfies EF ϕ.

Clauses 8–14 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.

Recall the transition system of Figure 3.3 (page 187) 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. $\mathcal{M}, s_0 \vDash p \land q$ 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.
3. $\mathcal{M}, s_0 \vDash \top$ holds by definition.
4. $\mathcal{M}, s_0 \vDash \text{EX}\ (q \land r)$ holds since we have the leftmost computation path $s_0 \to s_1 \to s_0 \to s_1 \to \ldots$ in Figure 3.5, whose second node s_1 contains q and r.
5. $\mathcal{M}, s_0 \vDash \neg\text{AX}\ (q \land r)$ holds since we have the rightmost computation path $s_0 \to s_2 \to s_2 \to s_2 \to \ldots$ in Figure 3.5, whose second node s_2 only contains r, but *not* q.

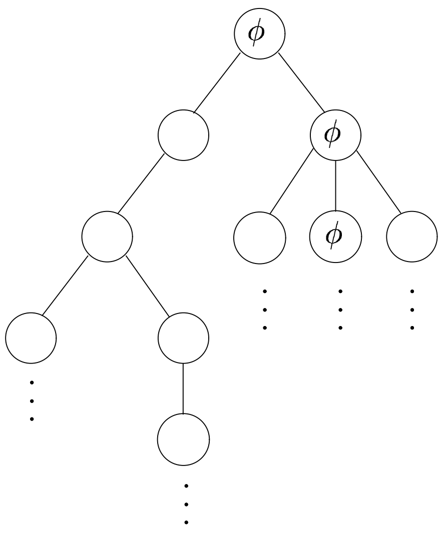


Fig. 3.20. A system whose starting state satisfies EG ϕ.

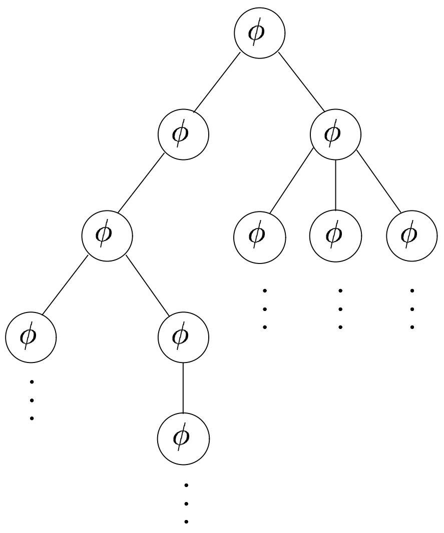


Fig. 3.21. A system whose starting state satisfies AG ϕ.

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 $s_2 \to s_2 \to s_2 \to \ldots$ beginning in s_2 such that r holds in all future states of

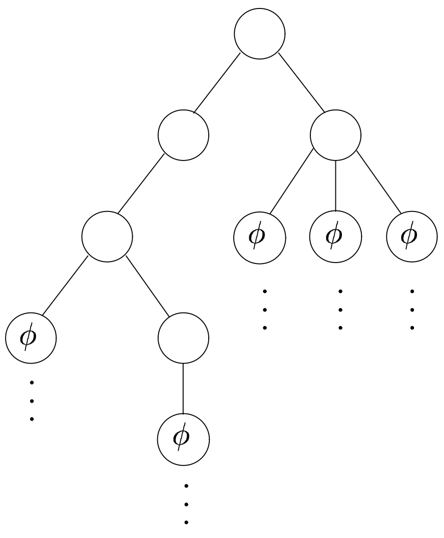


Fig. 3.22. A system whose starting state satisfies AF ϕ.

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 $s_0 \to s_2 \to s_2 \to s_2 \to \dots$ 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. $\mathcal{M}, s_0 \vDash \mathrm{AG}\ (p \vee q \vee r \to \mathrm{EF\ EG}\ r)$ 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: EF (started $\wedge$ $\neg$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:
  AG (requested $\rightarrow$ AF 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 AG AF enabled $\rightarrow$ AG AF 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:
  AG (AF enabled).
- Whatever happens, a certain process will eventually be permanently deadlocked:
  AF (AG deadlock).
- From any state it is possible to get to a restart state:
  AG (EF restart).
- An upwards travelling elevator at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:
  AG (floor=2 $\wedge$ direction=up $\wedge$ ButtonPressed5 $\rightarrow$
  A[direction=up U floor=5])
  Here, our atomic descriptions are boolean expressions built from system variables, e.g. floor=2.
- The elevator can remain idle on the third floor with its doors closed:
  AG (floor=3 $\wedge$ idle $\wedge$ door=closed $\rightarrow$ EG (floor=3 $\wedge$ idle $\wedge$ door=closed)).
- 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 AG $(n_1 \rightarrow \mathrm{EX}\, t_1)$ 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: 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

$$\mathrm{AF}\,\phi \equiv \mathrm{A}[\top\ \mathrm{U}\ \phi] \qquad\qquad \mathrm{EF}\,\phi \equiv \mathrm{E}[\top\ \mathrm{U}\ \phi]$$

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 $\top$.

*3.5.0.1 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

$$\alpha ::= (\phi \mathrel{\mathrm{U}} \phi) \mid (\mathrm{G}\, \phi) \mid (\mathrm{F}\, \phi) \mid (\mathrm{X}\, \phi)$$

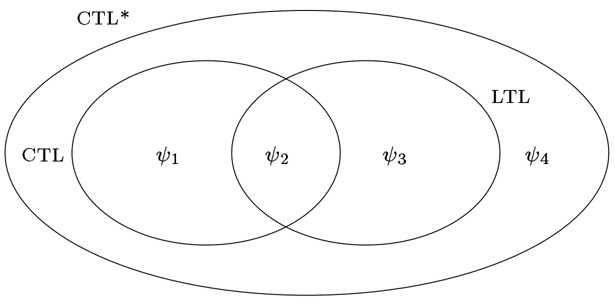
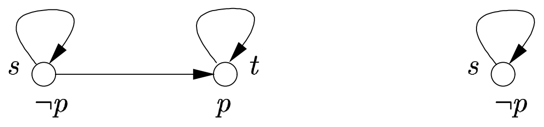


Fig. 3.23. The expressive powers of CTL, LTL and CTL*.

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 shown:

In CTL but not in LTL: $\psi_1 \stackrel{\text{def}}{=} \mathrm{AG\,EF}\,p$. 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: $\psi_4 \stackrel{\text{def}}{=} \mathrm{E}[\mathrm{G\,F}\,p]$, saying that there is a path with infinitely many p.

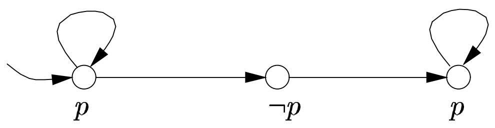
The proof that this is not expressible in CTL is quite complex and may be found in the papers co-authored by E. A. Emerson with others, given in the references. (Why is it not expressible in LTL?)

In LTL but not in CTL: $\psi_3 \stackrel{\text{def}}{=} \mathrm{A}[\mathrm{G\,F}\,p \to \mathrm{F}\,q]$, 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: $\psi_2 \stackrel{\text{def}}{=} \mathrm{AG}\,(p \to \mathrm{AF}\, q)$ 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:

$$\begin{aligned}\mathrm{E}[(p_1 \;\mathrm{U}\; q_1) \wedge (p_2 \;\mathrm{U}\; q_2)] \equiv \quad & \mathrm{E}[(p_1 \wedge p_2) \;\mathrm{U}\; (q_1 \wedge \mathrm{E}[p_2 \;\mathrm{U}\; q_2])] \\ & \vee \mathrm{E}[(p_1 \wedge p_2) \;\mathrm{U}\; (q_2 \wedge \mathrm{E}[p_1 \;\mathrm{U}\; q_1])]\ .\end{aligned}$$

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.

3.6.1.1 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 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.

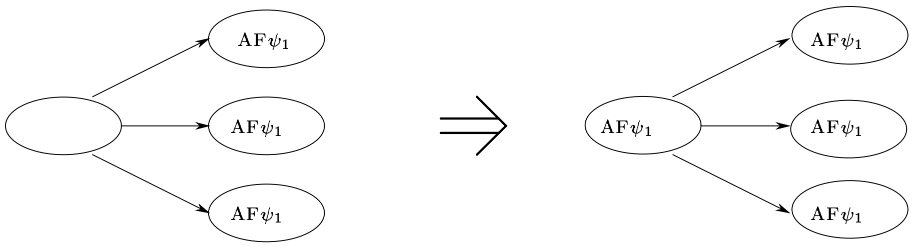
Suppose ψ is a subformula of ϕ and states satisfying all the *immediate* subformulas of ψ have already been labelled. We determine by a case analysis which states to label with ψ. If ψ is

- $\bot$: 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.

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

Repeat ...



... until no change.

Fig. 3.24. The iteration step of the procedure for labelling states with subformulas of the form AF ψ_1.

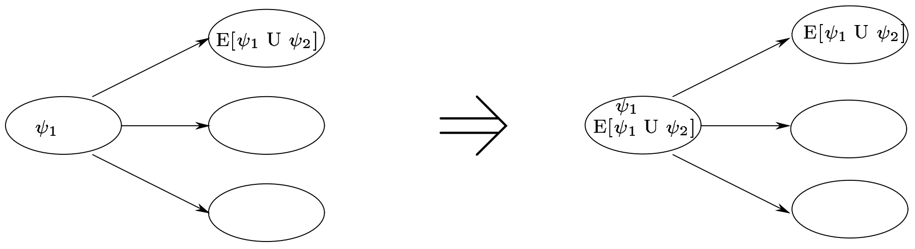


Fig. 3.25. The iteration step of the procedure for labelling states with subformulas of the form E[ψ_1 U ψ_2].

the number of transitions; the algorithm is linear in the size of the formula and quadratic in the size of the model.

3.6.1.2 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 EG ψ_1 *directly*:

- EG ψ_1:
  - Label *all* the states with EG ψ_1.
  - If any state s is *not* labelled with ψ_1, *delete* the label EG ψ_1.
  - Repeat: *delete* the label EG ψ_1 from any state if *none* of its successors is labelled with EG ψ_1; until there is no change.

Here, we label all the states with the subformula EG ψ_1 and then whittle down this labelled set, instead of building it up from nothing as we did in

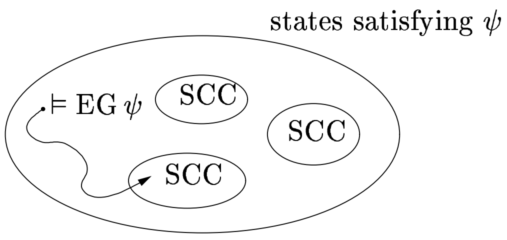


Fig. 3.26. A better way of handling EG.

the case for EU. Actually, there is no real difference between this procedure for EG ψ and what you would do if you translated it into $\neg$AF $\neg\psi$ as far as the final result is concerned.

3.6.1.3 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 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 E$[\neg c_2$ U $c_1]$; see Figure 3.27. The algorithm labels all states which satisfy c_1 during phase 1 with E$[\neg c_2$ 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.

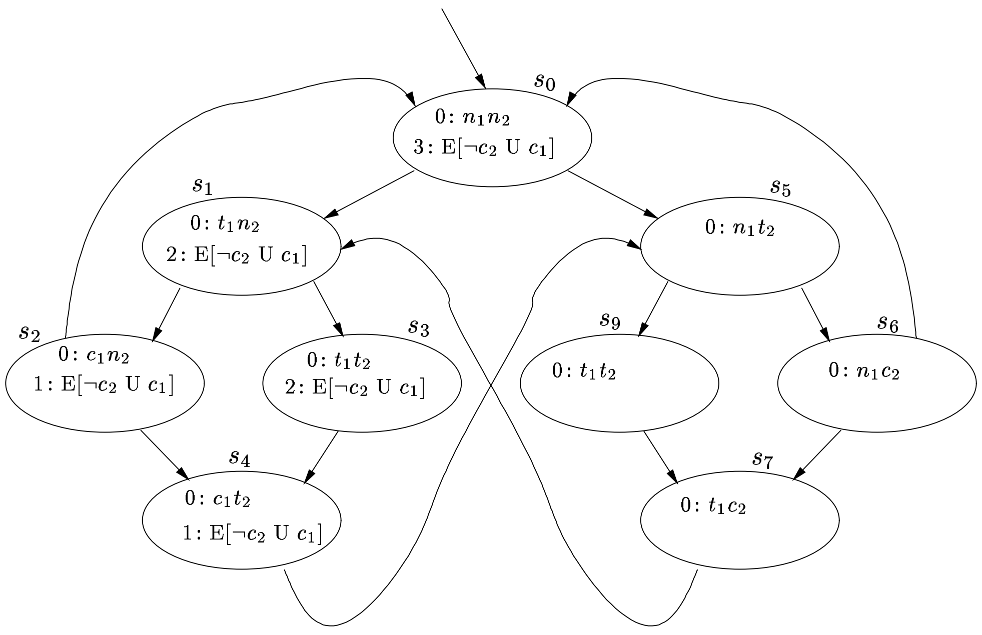


Fig. 3.27. An example run of the labelling algorithm in our second model of mutual exclusion applied to the formula E[$\neg c_2$ U c_1].

3.6.1.4 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 X ($\top \wedge$ EF p_3), since this is not a CTL formula.

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

```
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 $\phi_1 \to \phi_2$ : return SAT ($\neg\phi \vee \phi_2$)
        $\phi$ is AX $\phi_1$ : return SAT ($\neg$EX $\neg\phi_1$)
        $\phi$ is EX $\phi_1$ : return SAT$_{EX}$($\phi_1$)
        $\phi$ is A[$\phi_1$ U $\phi_2$] : return SAT($\neg$(E[$\neg\phi_2$ U ($\neg\phi_1 \wedge \neg\phi_2$)] $\vee$ EG $\neg\phi_2$))
        $\phi$ is E[$\phi_1$ U $\phi_2$] : return SAT$_{EU}$($\phi_1, \phi_2$)
        $\phi$ is EF $\phi_1$ : return SAT (E($\top$ U $\phi_1$))
        $\phi$ is EG $\phi_1$ : return SAT($\neg$AF $\neg\phi_1$)
        $\phi$ is AF $\phi_1$ : return SAT$_{AF}$ ($\phi_1$)
        $\phi$ is AG $\phi_1$ : return SAT ($\neg$EF $\neg\phi_1$)
    end case
end function
```

Fig. 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$_{EX}$, SAT$_{EU}$ and SAT$_{AF}$, respectively, if EX , EU or AF is the root of the input's parse tree.

```
function SAT$_{EX}$ ($\phi$)
 /* determines the set of states satisfying EX $\phi$ */
local var $X, Y$
begin
    $X$ := SAT ($\phi$);
    $Y$ := pre$_\exists$($X$);
    return $Y$
end
```

Fig. 3.29. The function SAT$_{EX}$. It computes the states satisfying ϕ by calling SAT. Then, it looks backwards along $\to$ to find the states satisfying EX ϕ.

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, $\to$ 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.

The algorithm is presented in Figure 3.28 and its subfunctions in Fig-

```
function SAT_AF (φ)
 /* determines the set of states satisfying AF φ */
local var X, Y
begin
    X := S;
    Y := SAT (φ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ pre_∀(Y)
    end
    return Y
end
```

Fig. 3.30. The function SAT$_{\mathtt{AF}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying AF ϕ in the manner described in the labelling algorithm.

```
function SAT_EU (φ, ψ)
 /* determines the set of states satisfying E[φ U ψ] */
local var W, X, Y
begin
    W := SAT (φ);
    X := S;
    Y := SAT (ψ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ (W ∩ pre_∃(Y))
    end
    return Y
end
```

Fig. 3.31. The function SAT$_{\mathtt{EU}}$. It computes the states satisfying ϕ by calling SAT. Then, it accumulates states satisfying E[ϕ U ψ] in the manner described in the labelling algorithm.

ures 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

$$\begin{aligned}\mathrm{pre}_\exists(Y) &= \{s \in S \mid \text{exists } s',\ (s \to s' \text{ and } s' \in Y)\}\\ \mathrm{pre}_\forall(Y) &= \{s \in S \mid \text{for all } s',\ (s \to s' \text{ implies } s' \in Y)\}\ .\end{aligned}$$

'Pre' denotes travelling backwards along the transition relation. Both functions compute a pre-image of a set of states. The function $\text{pre}_\exists$ (instrumental in $\texttt{SAT}_{\texttt{EX}}$ and $\texttt{SAT}_{\texttt{EU}}$) takes a subset Y of states and returns the set of states which *can* make a transition into Y. The function $\text{pre}_\forall$, used in $\texttt{SAT}_{\texttt{AF}}$, takes a set Y and returns the set of states which make transitions *only* into Y. Observe that $\text{pre}_\forall$ can be expressed in terms of complementation and $\text{pre}_\exists$, as follows:

$$\text{pre}_\forall(Y) = S - \text{pre}_\exists(S - Y) \tag{3.8}$$

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.

3.6.1.5 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.
- 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,

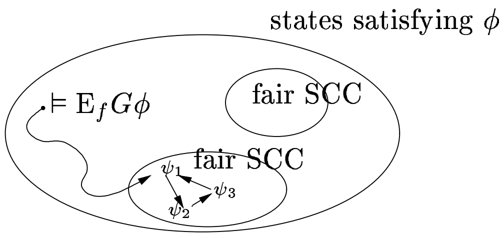


Fig. 3.32. Computing the states satisfying $\mathrm{E}_C\mathrm{G}\,\phi$. A state satisfies $\mathrm{E}_C\mathrm{G}\,\phi$ iff, in the graph resulting from the restriction to states satisfying ϕ, the state has a fair path from it. A fair path is one which leads to an SCC with a cycle passing through at least one state that satisfies each fairness constraint; in the example, C equals $\{\psi_1, \psi_2, \psi_3\}$.

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.

3.6.3.1 The basic strategy

Let $\mathcal{M} = (S, \to, L)$ be a model, $s \in S$, and ϕ an LTL formula. We determine whether $\mathcal{M}, s \models \phi$, i.e. whether ϕ is satisfied along all paths of $\mathcal{M}$ starting

```
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;
```

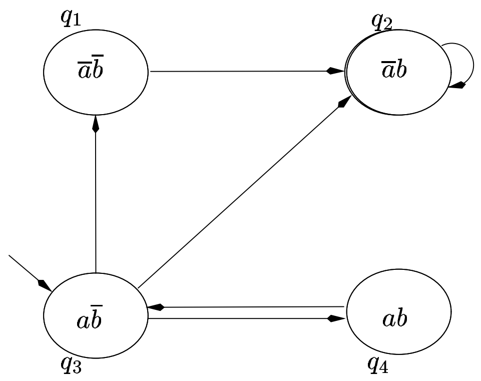


Fig. 3.33. An SMV program and its model $\mathcal{M}$.

at s. Almost all LTL model checking algorithms proceed along the following three steps.

Step 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.

Step 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.

Step 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 $q_3, q_2, q_2 \ldots$ does not satisfy it) we expect the model check to fail.

In accordance with Step 1, we construct an automaton $A_{a \mathrm{U} b}$ which characterises precisely the traces which satisfy a U b. (We use the fact that $\neg\neg(a$ U $b)$ is equivalent to a 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.

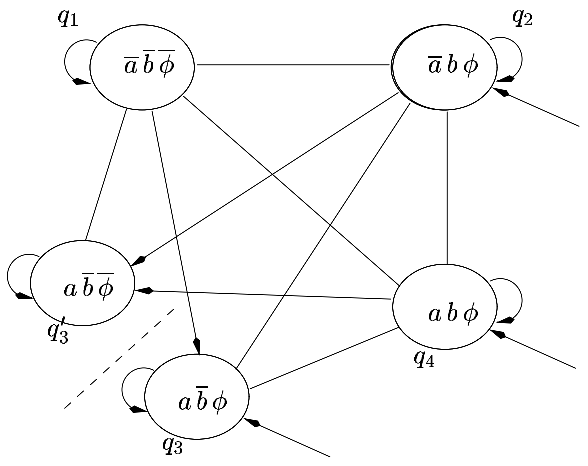


Fig. 3.34. Automaton accepting precisely traces satisfying $\phi \overset{\text{def}}{=} a$ U b. The transitions with no arrows can be taken in either direction. The acceptance condition is that the path of the automaton cannot loop indefinitely through q_3.

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;
- the path respects a certain 'accepting condition.' For the automaton of Figure 3.34, the accepting condition is that the path should not end $q_3, q_3, q_3 \ldots$, indefinitely.

For example, suppose t is $a\,\overline{b}, a\,\overline{b}, a\,\overline{b}, a\,b, a\,b, \overline{a}\,\overline{b}, a\,\overline{b}, a\,\overline{b}, \ldots$, eventually repeating forevermore the state $a\,\overline{b}$. Then we choose the path $q_3, q_3, q_3, q_4, q_4, q_1, q_3', q_3' \ldots$. 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.

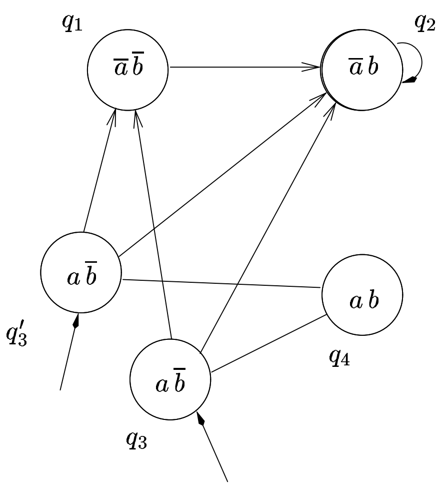
Why does the automaton of Figure 3.34 work as intended? To understand it, observe that it has enough states to distinguish the values of the propositions – that is, a state for each of the valuations $\{\overline{a}\,\overline{b}, \overline{a}\,b, a\,\overline{b}, a\,b\}$, and in fact two states for 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 $\phi \overset{\text{def}}{=} a \text{ U } b$ 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 $A_{a\text{U}b}$, we proceed to step 2. To combine the automaton $A_{a\text{U}b}$ 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 $A_{a\text{U}b}$ and select the transitions common to both, obtaining the combined system of Figure 3.35(right).

Step 3 now asks whether there is a 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 to Step 3, and as we expected, $\neg(a \text{ U } b)$ is not satisfied in all paths of the original system $\mathcal{M}$.

3.6.3.2 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



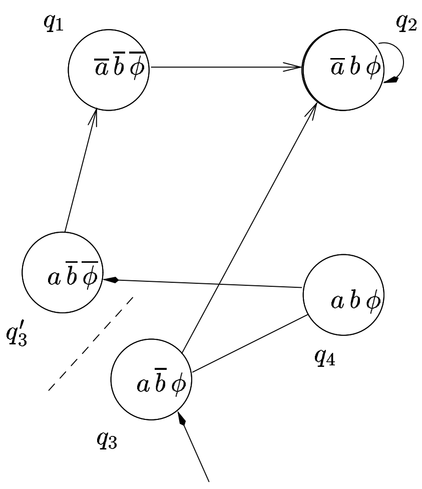


Fig. 3.35. Left: the system $\mathcal{M}$ of Figure 3.33, redrawn with an expanded state space; right: the expanded $\mathcal{M}$ and $A_{a \mathrm{U} b}$ combined.

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, $\mathcal{C}(a \text{ U } b) = \{a, b, \neg a, \neg b, a \text{ U } b, \neg(a \text{ U } b)\}$. 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 $\psi_1 \text{ U } \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\neg(\psi_1 \text{ U } \psi_2) \in q$, 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 $\psi_1 \text{ U } \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1 \text{ U } \psi_2 \in q'$.
- If $\neg(\psi_1 \text{ U } \psi_2) \in q$ and $\psi_1 \in q$ then $\neg(\psi_1 \text{ U } \psi_2) \in q'$.

These last two conditions are justified by the recursion laws

$$\begin{aligned}\psi_1 \mathrm{~U~} \psi_2 &= \psi_2 \lor (\psi_1 \land \mathrm{X}\,(\psi_1 \mathrm{~U~} \psi_2)) \\ \neg(\psi_1 \mathrm{~U~} \psi_2) &= \neg\psi_2 \land (\neg\psi_1 \lor \mathrm{X}\,\neg(\psi_1 \mathrm{~U~} \psi_2))\ .\end{aligned}$$

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 $\phi \mathrm{~U~} \psi$, namely that A_ϕ cannot stay for ever in states satisfying ϕ without ever obtaining ψ. 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 $q_3, q_3, \dots$.

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 $\chi_1 \mathrm{~U~} \psi_1, \dots, \chi_k \mathrm{~U~} \psi_k$ 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 $1 \le i \le k$, the run has infinitely many states satisfying $\neg(\chi_i \mathrm{~U~} \psi_i) \lor \psi_i$. 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 $\neg(\chi_i \mathrm{~U~} \psi_i) \lor \psi_i$. Let us advance through all those finitely many states, taking the suffix of the run none of whose states satisfies $\neg(\chi_i \mathrm{~U~} \psi_i) \lor \psi_i$, i.e. all of whose states satisfy $(\chi_i \mathrm{~U~} \psi_i) \land \neg\psi_i$. 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 $(p \mathrm{~U~} q) \lor (\neg p \mathrm{~U~} q)$. 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 $\neg p \mathrm{~U~} q$.

3.6.3.3 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 $\mathcal{M}$, 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 $\mathrm{EG}\,\top$. Thus, we ask the question: does the combined system have a path. The acceptance conditions of $A_{\neg\phi}$

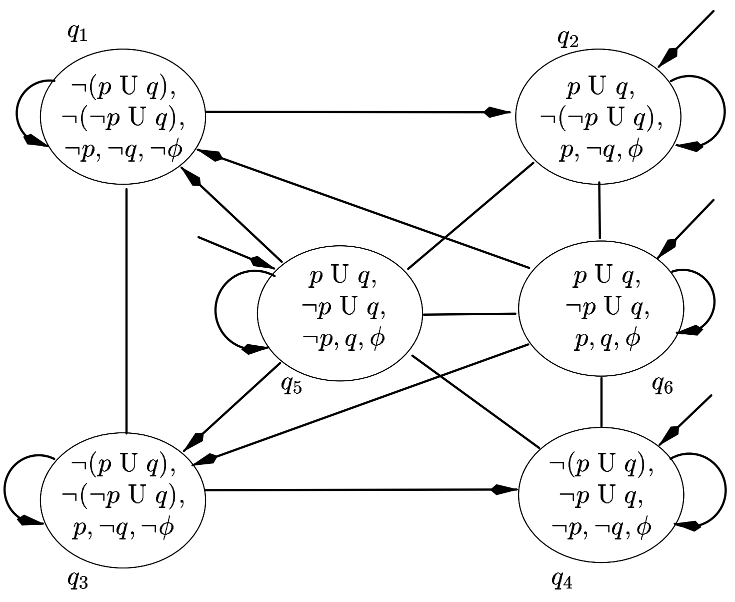

q_1
$\neg(p \mathrm{~U~} q)$, $\neg(\neg p \mathrm{~U~} q)$, $\neg p, \neg q, \neg\phi$
q_2
$p \mathrm{~U~} q$, $\neg(\neg p \mathrm{~U~} q)$, $p, \neg q, \phi$
q_5
$p \mathrm{~U~} q$, $\neg p \mathrm{~U~} q$, $\neg p, q, \phi$
q_6
$p \mathrm{~U~} q$, $\neg p \mathrm{~U~} q$, p, q, ϕ
q_3
$\neg(p \mathrm{~U~} q)$, $\neg(\neg p \mathrm{~U~} q)$, $p, \neg q, \neg\phi$
q_4
$\neg(p \mathrm{~U~} q)$, $\neg p \mathrm{~U~} q$, $\neg p, \neg q, \phi$

Fig. 3.36. Automaton accepting precisely traces satisfying $\phi \stackrel{\text{def}}{=} (p \mathrm{~U~} q) \vee (\neg p \mathrm{~U~} q)$. 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 $\{q_1, q_2, q_3, q_5, q_6\}$.

are represented as implicit fairness conditions for the CTL model checking procedure. Explicitly, this amounts to asserting 'FAIRNESS $\neg(\chi \mathrm{~U~} \psi) \vee \psi$' for each formula $\chi \mathrm{~U~} \psi$ occurring in $\mathcal{C}(\phi)$.

3.7 The fixed-point characterisation of CTL

On page 236, 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 $\mathcal{M}, s' \vDash \psi$.

```
function SAT$_{\mathrm{EG}}$ ($\phi$)
/* determines the set of states satisfying EG $\phi$ */
local var $X, Y$
begin
    $Y := \mathtt{SAT}(\phi)$;
    $X := \emptyset$;
    repeat until $X = Y$
    begin
        $X := Y$;
        $Y := Y \cap \mathrm{pre}_{\exists}(Y)$
    end
    return $Y$
end
```

Fig. 3.37. The pseudo-code for $\mathtt{SAT}_{\mathrm{EG}}$.

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 pseudo-code 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 $\phi_1 \to \phi_2$.

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 based on the intuitions given in Section 3.6.1.2: 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 EG ϕ says that $s_0 \vDash \mathrm{EG}\,\phi$ holds iff there exists a computation path $s_0 \to s_1 \to s_2 \to \dots$ such that $s_i \vDash \phi$ holds *for all* $i \geq 0$. We could instead express it as follows: EG ϕ holds if ϕ holds and EG ϕ 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 EG ϕ, because we need to know EG ϕ in

[1] Section 3.6.1.4 handles EG ϕ by translating it into $\neg\mathrm{AF}\,\neg\phi$, but we already noted in Section 3.6.1.2 that EG could be handled directly.

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.

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 $S \overset{\text{def}}{=} \{s_0, s_1\}$ and $F(Y) \overset{\text{def}}{=} Y \cup \{s_0\}$ 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

$$G(Y) \overset{\text{def}}{=} \texttt{if } Y = \{s_0\} \texttt{ then } \{s_1\} \texttt{ else } \{s_0\}\ .$$

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 $\texttt{SAT}_{\texttt{EU}}$ and $\texttt{SAT}_{\texttt{AF}}$ 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 263). 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 $\texttt{SAT}_{\texttt{EG}}$

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.

$\square$

Now we can see that the procedure $\texttt{SAT}_{\texttt{EG}}$ is correctly coded and terminates. First, note that the line $Y := Y \cap \mathrm{pre}_\exists(Y)$ in the procedure $\texttt{SAT}_{\texttt{EG}}$

(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 $\mathtt{SAT}(\phi)$[1]. 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 $[\![\mathrm{E}[\phi \mathrel{\mathrm{U}} \psi]]\!] = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists[\![\mathrm{E}[\phi \mathrel{\mathrm{U}} \psi]]\!])$. That tells us that $[\![\mathrm{E}[\phi \mathrel{\mathrm{U}} \psi]]\!]$ is a fixed point of the function $G(X) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists(X))$. As before, we can prove that this function is monotone. It turns out that $[\![\mathrm{E}[\phi \mathrel{\mathrm{U}} \psi]]\!]$ 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, $[\![\mathrm{E}(\phi \mathrel{\mathrm{U}} \psi)]\!]$ is the least fixed-point of G, and we have $[\![\mathrm{E}(\phi \mathrel{\mathrm{U}} \psi)]\!] = G^{n+1}(\emptyset)$.

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

[1] If you are sceptical, try computing the values $Y_0, Y_1, Y_2, \ldots$, 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.

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 $\emptyset$: $G^1(\emptyset) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathrm{pre}_\exists([\![\phi]\!])) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \emptyset) = [\![\psi]\!] \cup \emptyset = [\![\psi]\!]$, 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 $\mathtt{SAT}_{\mathtt{EU}}$ follows similarly to that of $\mathtt{SAT}_{\mathtt{EG}}$. 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 $\mathtt{SAT}_{\mathtt{EU}}$ 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 $[\![\mathrm{E}(\top \mathrm{~U~} p)]\!] = [\![\mathrm{EF}\, p]\!] = [\![\mathrm{EF}\, p]\!]$.

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,

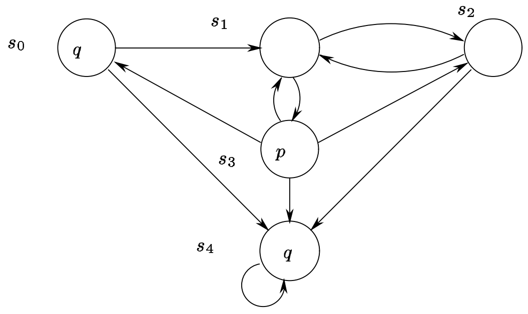


Fig. 3.38. A system for which we compute invariants.

where $\phi \overset{\text{def}}{=} q$. From Figure 3.38 we see that $[\![q]\!] = \{s_0, s_4\}$ and so $F(X) = [\![q]\!] \cap \text{pre}_\exists(X) = \{s_0, s_4\} \cap \text{pre}_\exists(X)$. Since $[\![\text{EG}\, q]\!]$ equals the greatest fixed point of F, we need to iterate F on S until this process stabilises. First, $F^1(S) = \{s_0, s_4\} \cap \text{pre}_\exists(S) = \{s_0, s_4\} \cap S$ since every s has some s' with $s \to s'$. Thus, $F^1(S) = \{s_0, s_4\}$.

Second, $F^2(S) = F(F^1(S)) = \{s_0, s_4\} \cap \text{pre}_\exists(\{s_0, s_4\}) = \{s_0, s_4\}$. Therefore, $\{s_0, s_4\}$ is the greatest fixed point of F, which equals $[\![\text{EG}\, q]\!]$ 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 analyzer according to the classification criteria for formal methods proposed on page 180.
2. Visit and browse the web sites [1] and [2] to find formal methods that interest you for whatever reason. Then classify them according to the criteria from page 180.

Exercises 3.2

1. Draw parse trees for the LTL formulas:
   (a) $\text{F}\, p \wedge \text{G}\, q \to p \;\text{W}\; r$
   (b) $\text{F}\,(p \to \text{G}\, r) \vee \neg q \;\text{U}\; p$
   (c) $p \;\text{W}\; (q \;\text{W}\; r)$
   (d) $\text{G}\,\text{F}\, p \to \text{F}\,(q \vee s)$
2. Consider the system of Figure 3.39. For each of the formulas ϕ:

[1] www.afm.sbu.ac.uk
[2] www.cs.indiana.edu/formal-methods-education/

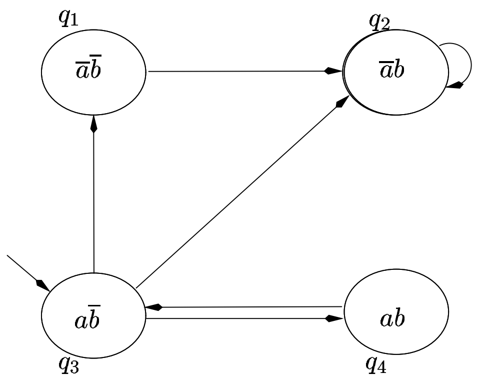


Fig. 3.39. A model $\mathcal{M}$.

(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 184), 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 194.
8. Recall the algorithm `NNF` on page 64 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 201).
   * * (a) Verify that `G(req -> F busy)` holds in all initial states.
   * (b) Does $\neg($`req` U $\neg$`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 199.
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) $\mathrm{F}\,\mathrm{G}\, r$

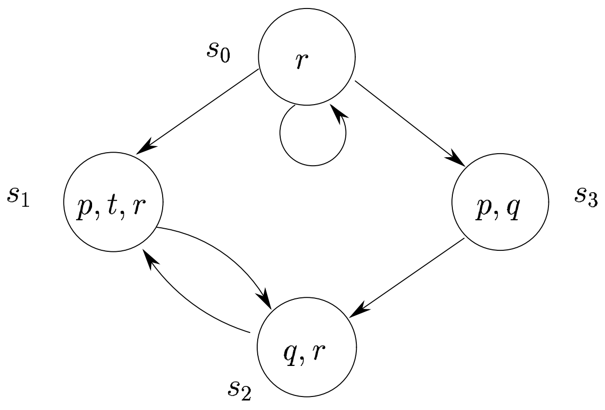


Fig. 3.40. A model with four states.

(b) $\mathrm{X\,X}\,r$
(c) $\mathrm{A}\neg\mathrm{G}\,\neg p$
(d) $\mathrm{F}\,[r\ \mathrm{U}\ q]$
(e) $\mathrm{EX\,X}\,r$
* (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.
   (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 $\mathrm{E}[\texttt{req}\ \mathrm{U}\ \neg\texttt{busy}]$ hold in all initial states of the model in Figure 3.9 on page 201?

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:

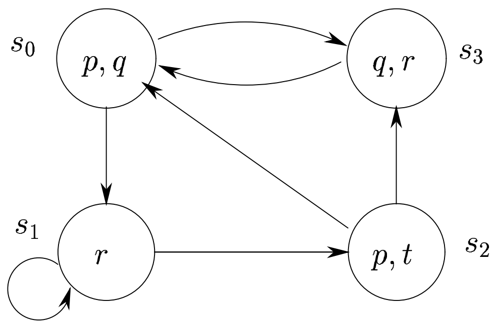


Fig. 3.41. Another model with four states.

* (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 220.

* (a) $[\![\top]\!] = S$,
(b) $[\![\bot]\!] = \emptyset$
(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, 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 217 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) $\mathrm{A}[\phi_1 \mathrel{\mathrm{U}} \mathrm{A}[\phi_2 \mathrel{\mathrm{U}} \phi_3]]$ and $\mathrm{A}[\mathrm{A}[\phi_1 \mathrel{\mathrm{U}} \phi_2] \mathrel{\mathrm{U}} \phi_3]$, 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 ? marks, 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 226.

13. Prove the equivalences (3.6) on page 225.

* 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 192 and 223 capture the stated 'informal' properties expressed in plain English.
3. Consider the set of LTL/CTL formulas $\mathcal{F} = \{\mathrm{F}\, p \rightarrow \mathrm{F}\, q, \mathrm{AF}\ p \rightarrow \mathrm{AF}\ q, \mathrm{AG}\ (p \rightarrow \mathrm{AF}\ q)\}$.
   - (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 $\mathrm{AG}\,(p \rightarrow \mathrm{AF}\,(s \wedge \mathrm{AX}\,(\mathrm{AF}\, t)))$. 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 64 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 227.
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) $\mathrm{A}[\mathrm{F}\, p \rightarrow \mathrm{F}\, q]$.

8. The aim of this exercise is to demonstrate the expansion given for AW at the end of the last section, i.e. $\mathrm{A}[p \mathrm{~W~} q] \equiv \neg E[\neg q \mathrm{~U~} \neg(p \vee q)]$.
   (a) Show that the following LTL formulas are valid (i.e. true in any state of any model):
      (i) $\neg q \mathrm{~U~} (\neg p \wedge \neg q) \rightarrow \neg \mathrm{G}\, p$
      (ii) $\mathrm{G}\, \neg q \wedge \mathrm{F}\, \neg p \rightarrow \neg q \mathrm{~U~} (\neg p \wedge \neg q)$.
   (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 208. 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 205).

* 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 197).

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 199).

5. Prove that (3.8) on page 238 holds in all models. Does your proof require that for every state s there is some state s' with $s \rightarrow s'$?

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 236 by writing a special procedure for AG ψ_1, without rewriting it in terms of other formulas[1].

* 8. Write the pseudo-code for $\mathtt{SAT}_{\mathrm{EG}}$, based on the description in terms of deleting labels given in Section 3.6.1.2.

* 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 $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ and the n fairness constraints $\phi_1, \phi_2, \dots, \phi_n$? 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 205.

Exercises 3.7

1. Consider the functions

$$H_1, H_2, H_3\colon \mathcal{P}(\{1,2,3,4,5,6,7,8,9,10\}) \to \mathcal{P}(\{1,2,3,4,5,6,7,8,9,10\})$$

[1] Question: will your routine be more like the routine for AF, or more like that for EG on page 233? Why?

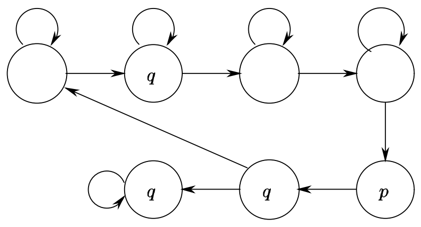


Fig. 3.42. Another system for which we compute invariants.

defined by

$$\begin{aligned} H_1(Y) &\stackrel{\text{def}}{=} Y - \{1,4,7\} \\ H_2(Y) &\stackrel{\text{def}}{=} \{2,5,9\} - Y \\ H_3(Y) &\stackrel{\text{def}}{=} \{1,2,3,4,5\} \cap (\{2,4,8\} \cup Y) \end{aligned}$$

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 $i = 1, 2, \ldots$ and Theorem 3.24.

(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) $F_1\colon \mathcal{P}(S) \to \mathcal{P}(S)$ with $F_1(Y) \stackrel{\text{def}}{=} A \cap F(Y)$ is monotone

(b) $F_2\colon \mathcal{P}(S) \to \mathcal{P}(S)$ 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[1]. NuSMV[2] is a reimplementation, developed in Trento by A. Cimatti, and M. Roveri and is aimed at being customisable and extensible. Extensive documentation

[1] `www.cs.cmu.edu/~modelcheck/`
[2] `nusmv.irst.itc.it`

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[3] 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 WWW site 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[1].

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[2]. A model checker called FDR2 based on the process algebra CSP is available[3]. The Edinburgh Concurrency Workbench[4] and the Concurrency Workbench of North Carolina[5] 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[6].

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/`.

[3] `www-cad.eecs.berkeley.edu/~kenmcmil/`
[1] `www.cis.ksu.edu/~dwyer/spec-patterns.html`
[2] `netlib.bell-labs.com/netlib/spin/whatispin.html`
[3] `www.formal.demon.co.uk/FDR2.html`
[4] `www.dcs.ed.ac.uk/home/cwb`
[5] `www.csc.ncsu.edu/eos/users/r/rance/WWW/cwb-nc.html`
[6] `http://bogor.projects.cis.ksu.edu/`

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] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan 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] George 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] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe 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] Alan Martin. Adequate sets of temporal connectives in CTL. *Electronic Notes in Theoretical Computer Science*, 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] Moshe 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, 211
  acknowledgement channel, 211
  alternating the control bit, 211
  fairness, 211
  main SMV program, 214
absorption laws, 92
abstract data type
  sets, 234
abstraction, 182, 237, 255
  and non-determinism, 199
accessibility relation, 321, 332, 349
adequate set of connectives
  for CTL, 224, 231, 239, 412
  for LTL, 194
  for propositional logic, 71, 91, 96
agent, 319, 331, 340
algebraic specification, 177
algorithm
  deterministic, 60
algorithm `apply`, 388
  complexity, 396
  control structure, 390
  recursive descent, 390
algorithm `CNF`, 60
algorithm `reduce`, 387
  complexity, 396
algorithm `restrict`, 391
  complexity, 396
algorithm `reduce`
  example execution, 388
Alloy
  [], 157
  `fun`-statement, 160
  `with`, 152
  assertion, 149
  check directive, 149
  consistency check, 149
  constrain signature, 155
  counterexample, 149
  dot operator, 149
  extending signature, 176
  implication, 152
  let-expression, 157
  limitations, 161
  modifiers, 155
  module, 150
  opening module, 161
  polymorphism, 161
  postcondition, 156
  precondition, 156
  reflexive, transitive closure, 176
  run directive, 150
  signature, 149
    instance, 149
  small scope hypothesis, 148, 176
  transitive closure, 161
  universal quantification, 152
alternating bit protocol, 211
always in the future, 331
and-elimination, 6, 353
and-introduction, 6, 353
application domain, 180, 266
approach
  model-based, 180
  proof-based, 180
approximants
  $\mu_m Z.f$, 408
  $\nu_m Z.f$, 409
arity, 103
array, 310
  bounds, 141, 298
  field, 298
  of integers, 298
  section, 298
artificial intelligence, 318
artificial language, 97
assertion checking, 148
assignment, 199
  initial, 302
  non-deterministic, 212, 238
  program notation, 270
  statement, 270
associativity laws, 57, 91
assumption, 4
  discharging, 28, 342
  temporary, 11, 126

asynchronous
  circuit, 373
  interleaving, 196
atomic formula, 348
  of modal logic, 319
  of predicate logic
    meaning, 129
axiom
  5, 344
  T, 355
  4, 340, 343, 353
  5, 343, 353
  T, 340, 343, 353, 357
  for assignment, 280
  for equality, 112
  instance, 281

Backus Naur form (BNF), 34
backwards breadth-first search, 233, 239
base case, 42, 43, 89
basic modal logic, 319
BDD, 379
  hi(n), 387
  lo(n), 387
  as boolean function, 380
  complement, 381
  consistent path, 380
  edge, 376
  examples, 380
  has an ordering, 382
  layer of variables, 376
  line
    dashed, 376, 379
    solid, 376, 379
  ordered, 382
  read-1, 422
  reduced, 380
  removal of duplicate non-terminals, 378
  removal of duplicate terminals, 378
  removal of redundant tests, 378
  satisfiable, 380
  subBDD, 378
  which is not a read-1-BDD, 393
  which is not an OBDD, 383
  with duplicated subBDDs, 378
belief, 331
binary decision diagram, 379
binary decision tree, 376
  redundancies in, 377
binding priorities, 183, 216
  for basic modal logic, 319
  for integer expressions, 270
  for KT45^n, 348
  for predicate logic, 105
  for propositional logic, 5
  for relational mu-calculus, 406
bit, 138
  control, 211
  least significant, 397
  most significant, 397
  one-bit channel, 212
  two-bit channel, 212
blocks of code, 270
Boole, G., 96, 389
boolean algebra, 19
boolean connective, 217, 322
boolean existential quantification, 392
boolean expression, 270, 282
boolean forall quantification, 395
boolean formula
  independent of a variable, 390
  semantically equivalent, 389
  truth table, 374
boolean function
  'don't care' conditions, 420
  as a binary decision tree, 376
  symbolic representation, 373
boolean guard, 293
boolean variable, 373
bottom-elimination, 21
bottom-introduction (see 'not-elimination'), 21
box-elimination, 342
box-introduction, 342
branching-time logic, 181

case
  overlap, 63
case analysis, 62, 63, 117
case-statement, 18, 200
characteristic function, 399
Church, A., 138
circuit
  2-bit comparator, 425
  asynchronous, 202, 405
  sequential, 374
  synchronous, 202, 373, 404, 429
circular definition, 225
Clarke, E., 263
classical logic, 30, 341
client, 268
clock tick, 198
CNF, 56
code
  specification, 266
  verification, 266
coding
  AF, 411
  EF, 410
  EG, 411
  EU, 411
  EX, 410
  examples of symbolic evaluation, 411
  fair EG, 413
  fair EU, 413
  fair EX, 413
  set of fair states, 413
command, 270
  atomic, 270
  compound, 271
common knowledge, 345, 349
  as invariant, 406
communicating processes, 265

communication protocol, 202
Compactness Theorem, 142
completeness
  of natural deduction for predicate logic, 100
  of natural deduction for propositional logic, 55
complexity
  exponential, 237
  of `apply`, 421
  of brute force minimal-sum section algorithm, 299
  of fairness, 413
  of labelling algorithm, 231, 233
  of labelling EG_C, 239
composition
  sequential, 289
  synchronous, 202
compositional semantics, 39
compositionality
  in model checking, 237
computability, 136
computation
  intractable, 50
computation path, 187, 220
  fair, 239
computation trace, 295
computation tree logic, 182
computation tree logic, 318, 331
computational behaviour, 318
computer program, 106
concatenation, 130, 137
conclusion, 4, 283
concurrency, 266
conjunct, 57
conjunction, 4, 301
  infinite, 349
connective
  adequate set, 235
  unary, 183, 216
consistency, 320, 329
consistency checking, 148
constant symbol, 162
constraints
  inconsistent, 76
  SAT solver, 71
contradiction, 20, 122, 331, 337
control structure, 270, 271
controlling value, 421
copy rule, 20, 342
core programming language, 269, 298
correspondence theory, 338
counter example, 128, 135, 329, 369
counter trace, 181
critical section, 195
CTL, 182, 263, 318, 331
  as a subset of CTL*, 226
  expressive power, 228
  modalities, 318
  with boolean combinations of path formulas, 228, 259
CTL connectives
  fair, 413
CTL formula
  square brackets, 217
CTL*, 263

DAG, 73
dag, 379
dashed box
  flavour, 353
data structure, 128
de Morgan laws, 59, 224, 260
  for modalities, 326
deadlock, 186, 191, 223, 227
debugging systems, 230
debugging systems, 266
decision problem, 136
  of validity in predicate logic, 138
decision procedure, 56
declarative explanation, 26
declarative sentence, 2, 97
  truth value, 37
default case, 200
definition
  inductive, 33
description
  informal, 267
  language, 179, 181
Dijkstra, E., 294
directed graph, 141, 185, 379
  acyclic, 71, 379
  cycle, 379
disjunction, 4
  of literals, 56, 58
distributivity laws
  of box modality, 326
  of F connective, 192
  of propositional logic, 19, 61, 92
dividend, 313
domain assumption, 153
double negation-elimination, 367
double negation-introduction, 367

elimination rule, 6, 111
Emerson, E. A., 263
encoding, 133
entailment
  in program logics, 288
environment
  and non-determinism, 199
  for concurrent programs, 180
  for predicate logic formulas, 132
equality, 272
  intentional, 111
  program notation, 270
  structural, 158
  symbol, 111
equivalence relation, 334, 340, 353
equivalent formulas
  of basic modal logic, 327
  of CTL, 223–225
  of KT4, 341

  of KT45, 340
  of LTL, 192
  of predicate logic, 121
  of propositional logic, 16
  of relational mu-calculus, 427
exclusive-or, 397, 414
existential quantifier, 223
existential second-order logic, 144, 161
exists-elimination, 117
exists-introduction, 116

factorial
  of a natural number, 271
  program, 271, 295
fairness
  nested fixed points, 413
  symbolic model checking, 412
fairness constraint, 199, 203
  simple, 239, 261
FAIRNESS running, 212
Fibonacci numbers, 89
field index, 298
finite automata, 422
finite data structure, 230
first-order logic, 97
fixed point, 248
  greatest, 248, 249
  least, 248, 249
  semantics for CTL, 225, 246
flow of control, 271
Floyd, R., 279
for-statement, 310
forall-elimination, 113
forall-introduction, 114
formal
  path, 226
formula
  atomic, 182
  height, 44, 89
  Horn, 67
  ill-formed, 184
  immediate subformula, 231
  of basic modal logic, 326
  of CTL, 216
    atomic, 216
    ill-formed, 217
    well-formed, 216
  of LTL
    valid, 260
  of predicate logic, 104
  of propositional logic, 34, 50
    well-formed, 33, 44
  of relational mu-calculus, 406
  positive, 341, 357, 362
  scheme, 324, 330
    K, 327
    in propositional logic, 324
    instance, 324
  subformula, 35
frame, 335
free for x in ϕ, 109, 113

Frege, G., 177
function
  in predicate logic, 129
  monotone, 248
    a non-example, 248
  nullary, 103
  recursive, 258
  SAT, 234, 235
    termination, 263
  SATaf, 236
  SATag, 263
  SATeg, 261
  SATeu, 236
  SATex, 235
  symbol, 100, 102, 162
    binary, 102
  translate, 258
function $\mathrm{pre}_\forall(X)$, 236
function $\mathrm{pre}_\exists(X)$, 236, 401
function $\mathrm{pre}_\forall(X)$, 401
function SAT
  correctness, 248
future
  excludes the present, 258, 367
  includes the present, 189, 258, 367
  whether it includes the present, 331

G-reachable, 351
  in k steps, 351
Gödel, K., 100
Gentzen, G., 96
Global Assembly Cache, 154
grammar, 34
  clause, 279
guided simulation, 160

Halpern, J., 263
higher-order logic, 146
Hoare triple, 273
Hoare, C. A. R., 273, 279
Hodges, W., 177
Horn clause, 67, 144
hybrid rule, 357

if-statement, 291
implementation
  compliant, 148
implication, 4
  logical, 288
implies-elimination, 9
implies-introduction, 12
in-order representation, 36
inconsistency, 268
index, 137
induction
  course-of-values, 44
  hypothesis, 42, 43
  in model checking, 237
  mathematical, 41
inductive step, 42
infix notation, 130, 217

information
  negative, 357
input parameter, 62
integer
  expression, 269
integer label, 387
integer multiplication, 397
interface between logics, 287
interleaving
  formulas with code, 285
  transitions, 196, 202
introduction rules, 6, 111
introspection
  negative, 332, 340
  positive, 332, 340
intuitionistic logic, 30, 124, 341
invariants, 283
  discovering, 294
  SAT solver, 70
iterative squaring, 430

Jape, 178
justification, 286, 287, 342

Knaster-Tarski Theorem, 249
knowledge
  common, 346
  distributed, 349
  domain-specific, 106
  false, 334
  formula
    positive, 357
  idealised, 332, 340
  in a multi-agent system, 319
  modality, 348
  of agent, 319, 332
Kozen, D., 431
Kripke model, 174, 321
  as a counter example, 369
  for KT45^n, 350
Kripke, S., 321, 327

Löwenheim-Skolem Theorem, 143
label
  adding, 231
  deleting, 232
labelling
  AF, 231
  EG, 232
  EG$_C$, 239
  EU, 231
  EX, 231
labelling algorithm, 230
labelling function
  coding subsets, 399
  for Kripke model, 321
  for LTL model, 185
  frame does not have one, 335
language construct, 310
law of excluded middle, 25
LEM
  instance, 341
linear-time logic, 181
linear-time temporal logic, 183
Linear-time temporal logic (LTL), 182
literal, 56, 63
liveness, 198, 203
  property, 195, 197, 215, 238
logic engineering, 319, 328
logic programming, 50, 177
logical level, 288
logical variables
  of Hoare triple, 279
look-up table, 132, 145
  up-dated, 132
LTL, 183
LTLSPEC, 200, 212

machine state, 272
McMillan, K., 263
memoisation
  of computed OBDDs, 390
midcondition, 280, 286
minimal-sum section, 299
minimal-sum-section problem, 317
modal connective
  C_G, 349
  K_i, 348
modal logic, 318
  K, 339
  KT4, 340
  KT45, 339, 340
  normal, 339
  S4, 340
  S5, 339
modality, 320
  diamond, 320
  path, 228
model
  for propositional logic, 38
  of KT45^n, 349
  of basic modal logic, 321, 337
  of CTL, 322
    pictorial representation, 256, 257
  of intuitionistic propositional logic, 341
  of KT45, 350
  of KT45^n, 351
  of LTL
    pictorial representation, 186
  of predicate logic, 100, 129
  under-specified, 148
model checker, 181
model checking, 146, 180, 182, 265
  algorithm, 225, 234, 239, 331
  debugging, 410
  example, 190, 220
  with fairness constraints, 238
model-based verification, 179, 181
module, 275
modulo 8 counter, 426
modus ponens, 9
modus tollens, 10, 367

muddy-children puzzle, 356, 359
multiplicity constraint, 157
Mutex model
  pictorial representation, 196
mutual exclusion, 195

natural deduction
  extension to predicate logic, 99
  for modal logic, 345
  for temporal logic, 181
  inventor, 96
natural deduction rules
  for basic modal logic, 342
  for KT45^n, 354, 370
  for predicate logic, 111
  for propositional logic, 27
necessity
  logical, 320, 330
  physical, 330
negation, 4
negation-elimination (see 'bottom-elimination'), 21
negation-introduction, 22
nested boolean quantification, 410
network
  synchronous, 182
no strict sequencing, 195, 197, 223
node
  initial, 379
  leaf, 107
  non-terminal, 376
  shared, 72
  terminal, 376, 379
non-blocking protocol, 195, 197, 223
non-determinism, 198
non-termination, 271
normal form, 55, 56
  conjunctive, 56, 375
  disjunctive, 375
  negation, 61
    CTL*, 259
    LTL, 194, 255
  product-of-sums, 423
  sum-of-products, 421
not-elimination, 21
not-introduction, 22

OBDD, 382
  absence of redundant variables, 385
  canonical form, 384
  complementation, 401
  definition, 382
  extensions, 397
  for $\mathrm{pre}_\exists(X)$, 403
  for $\mathrm{pre}_\forall(X)$, 403
  integer multiplication, 397
  intersection, 401
  limitations, 397
  memoisation, 390
  nested boolean quantification, 396
  of a transition relation, 402
  of an even parity function, 385
  of the odd parity function, 416
  optimal ordering, 424
  reduced, 383
    unique representation, 383
  reduced one for logical 'iff', 416
  representing subsets, 399
  running time of algorithms
    upper bounds, 396
  sensitivity of size, 385
  synthesis of boolean formula, 396
  test
    for implication, 387
    for satisfiability, 387
    for semantic equivalence, 385
    for validity, 387
  union, 401
  variations, 397
odd parity function, 416
omniscience
  logical, 331, 332
or-elimination, 17
or-introduction, 17
overloading
  of $\vDash$, 134
  of proof rules, 111

parity function
  even, 384
    as OBDD, 385
parity OBDD, 397
parse tree
  for a predicate logic formula, 106
  of a term, 164
  of a basic modal logic formula, 319
  of a CTL formula, 217
  of propositional logic formula, 34
  root, 35
  subtree, 35
  underspecified, 324
partial correctness, 275
partial order reduction, 237
pattern
  `checkEU` (f, g), 413
  `checkEX` (f), 413
pattern matching, 6, 115, 290
place holder, 98
possibility, 328
  logical, 320
possible world
  semantics, 327
Post correspondence problem, 137
postcondition, 156
  in program logic, 273
Prawitz, D., 96
precondition, 156
  in program logic, 273
    weakest, 286
  of algorithm, 64
predicate, 97
  binary, 99

number of arguments, 99
unary, 99
predicate logic, 97
consistent set of formulas, 134
satisfiability, 134
semantic entailment, 134
validity, 134
prefix, 130
notation, 217
ordering, 130
premise, 280
Prior, A., 263
problem
instance, 137
reduction, 137
procedural interpretation, 26
process
concurrent, 195
instantiation, 213
processor, 266
program
behaviour, 274
bug, 266
code, 286
construct, 270
correctness, 64, 69, 234
derived, 310
diverging, 276
documentation, 266
environment, 268
finite-state, 373
fragment, 272
logic, 285
methodology, 268
procedures, 272
sequential, 265
termination, 69, 93, 249, 275
variable, 236, 277
verification, 279
formal, 269
program execution, 328, 332
programming by contract, 307
Eiffel, 307
programming language
imperative, 269
proof
box
for $\rightarrow$i, 11
for forall-introduction, 114
for modal logic, 342
opening, 28
side by side, 22
by contradiction, 25
calculus, 265, 269
construction, 279
constructive, 124
dashed box, 342, 353
fragment, 289
indirect, 30
of correctness, 247
of termination, 276
partial, 292
partial correctness, 279, 311
search, 50
solid box, 342
strategy, 119, 274
subproof, 282
tableaux, 279
theory, 97, 127, 181
total correctness, 303
proof rules, 5
for implication, 283
for assignment, 279
for conjunction, 6
for disjunction, 17
for double negation, 8
for equality, 112
for existential quantification, 116
for if-statements, 282, 291
modified, 292
for implication, 12, 287
for KT45^n, 353
for negation, 20
for quantifiers, 116
for sequential composition, 279, 285
for universal quantification, 113
for while-statements, 283, 293, 298
schema, 115
subformula property, 117
proof tableaux
complete, 302
proof-based verification, 179, 265
proposition, 2
propositional logic, 97
protocol, 195, 196
provability
undecidability of predicate logic, 141

quantifier, 322, 326
equivalences, 192
in predicate logic, 98
binding priorities, 105
equivalences, 134
meaning, 128
Quielle, J., 263

reachability, 141, 142
reasoning
about knowledge, 339, 345
constructive, 30
in an arbitrary accessible world, 342
informal, 357
quantitative, 268
unsound, 291
record
field, 201
recursion
mutual, 226
recursive call, 291
reductio ad absurdum, 25, 123
reduction to absurdity, 25
reflexive, transitive closure, 174

regular language, 422
relation
  binary, 185
  Euclidean, 333, 340
  functional, 333
  linear, 333
  reflexive, 145, 333, 336
    as formula, 113
  serial, 333, 368
  symmetric, 333
    as formula, 113
  total, 334
  transition, 185
  transitive, 145, 333, 337
    as formula, 113
relational mu-calculus
  fixed-point operators, 408
requirement
  informal, 267, 272, 299
requirements, 147
restriction, 389
right-associative, 5
root of a parse tree, 140
rule
  derived, 24
  hybrid, 10
Russell's paradox, 172

safety property, 195, 196, 213
SAT solver
  cubic, 77
  forcing rules, 73
  permanent marks, 77
  temporary marks, 76
satisfaction
  in a frame, 335
  in a frame for KT45^n, 351
satisfaction relation
  for relational mu-calculus, 407
  for basic modal logic, 322
  for KT45, 350
  for LTL, 187
  for partial correctness, 275
  for predicate logic, 132
  for relational mu-calculus, 407
  for total correctness, 275
satisfiability, 375
  3SAT, 423
  deciding, 67
  of a propositional logic formula, 88
  undecidability of predicate logic, 140
SCC
  fair, 239
scheduler
  fair, 205
scope
  of a dummy variable, 121
  of a variable, 107, 117
  of an assumption, 28, 117, 342
search space, 117, 138
second-order logic, 146
semantic entailment
  for predicate logic, 147
  for basic modal logic, 325
  for KT4, 341
  for normal modal logics, 339
  for predicate logic, 100
  for propositional logic, 46
  for relational mu-calculus, 427
semantic equivalence, 39
semantics
  of $\mu Z.f$, 408
  of $\nu Z.f$, 409
  of basic modal logic, 322
  of boolean quantification, 408
  of CTL, 218
  of EG, 247
  of equality, 136
  of predicate logic, 127
  of propositional logic, 39
  of relational mu-calculus, 407
  of Until, 188
sentence
  atomic, 4
  components, 97
  declarative, 97
  in predicate logic, 133
sequent, 5
  invalid, 120
Shannon expansion, 390
side condition, 112, 114
Sifakis, J., 263
small scope hypothesis, 148
SMV, 263
  main program for ABP, 214
  module, 201
    receiver, 213
    sender, 212
    for channel, 214
    instantiation, 201
  process, 405
  program
    example, 199
    for Mutex, 203
  specification, 200
software
  life-cycle, 147
  micromodel, 147
  reliability, 154
  requirements, 147
  specification, 147
  validation, 147
soundness
  of forall-elimination, 113
  of natural deduction
    basic modal logic, 369
    predicate logic, 100, 127
    propositional logic, 46
  of program logics, 277
  of proof rule for while-statements, 293
  of the substitution principle, 112
specification

for ABP, 213
formal, 268
informal, 268
language, 179
of a predicate, 162
patterns, 264
practical pattern, 191, 222
truth table, 59
specifications, 199
Spin, 264
state
critical, 196
explosion, 237
explosion problem, 264
fair, 413
final, 147
formula, 226
global, 196
graph, 187
initial, 147, 196, 230, 255, 261, 274
non-critical, 196
of a system, 278
of core program, 273
reachable, 255
resulting, 273, 310
space, 237
splitting states, 198
transition, 147
trying, 196
state machine, 147
storage
location, 299
state, 271
store
of core program, 273
string, 256, 319
binary, 130, 137
empty, 130
strongly connected component, 233
structural equality, 158
structural induction, 44, 52
subformula, 184
substitution
in predicate logic, 109
instance, 336
instance of tautology, 326
principle, 112
symbolic model checking, 398
syntactic
domain, 269, 270
syntax
of basic modal logic, 319
of boolean expressions, 270
of boolean formulas, 414
of CTL, 215, 216
of CTL*, 226
of KT45^n, 349
of LTL, 183
of predicate logic, 104
of propositional logic, 34
of relational mu-calculus, 406
of terms, 103
system
asynchronous, 264
interleaving model, 405
simultaneous model, 405
axiomatic, 96
commercially critical, 179, 266
component, 213
concurrent, 180
debugging, 181
description, 201
design, 181
development, 180
elevator, 191, 223
finite-state, 265
hybrid, 287
infinite-state, 265
mission-critical, 179
multi-agent, 345
physical, 182
reactive, 180, 266, 373
safety-critical, 179, 266
transition, 181
verification, 265

tautology, 50
temporal connective
AF, 219
AG, 219
AU, 219
AX, 219
EF, 219
EG, 219
EU, 219
EX, 219
temporal connectives, 183
temporal logic, 181, 318
term, 103
interpretation, 132
term-rewriting system, 178
termination
Collatz $3n + 1$, 306
proof, 276
tertium non datur, 25
theorem, 13
prover, 110, 141
proving, 178
time
continuous, 181
discrete, 181
top
marking, 68
total correctness, 275
transition relation, 185
for SMV programs, 404
transition system, 181
of ABP program, 255
of Mutex code, 207
of SMV program, 200
unwinding, 187, 220, 230
translation

English into predicate logic, 99, 105
tree
  infinite, 187, 220
truth
  dynamic, 181
  mode, 318, 320
  of knowledge, 339
  static, 181
  value
    for predicate logic, 132
    for propositional logic, 3
truth table
  for conjunction, 38
truth tables, 39
type, 12, 340
  checking, 12
  theory, 177

unary connective, 319
undecidability
  of provability, 141
  of satisfiability, 140
  of validity in predicate logic, 138
universal quantification, 278
universal quantifier, 223
universal second-order logic, 145, 161
universe of concrete values, 129
unreachability, 145
unsound sequent, 171
Until
  in natural language, 189
  negating, 194
updated valuation, 407

valid sequent
  of modal logic, 343
  partial correctness, 277
  total correctness, 277
validity
  in basic modal logic, 326
  in KT45^n, 352
  in propositional logic, 88
  undecidability in predicate logic, 138
valuation
  for propositional logic, 38
  in predicate logic, 127
  in relational mu-calculus, 407
value
  initial, 213, 278
Vardi, M., 263
variable, 98, 269
  boolean, 237, 255, 373
  bound, 107
  capture, 109
  dummy, 114
  free, 107
  local, 272
  logical, 278, 301
variable ordering
  compatible, 383
  list, 382
variant, 304
verification
  full, 180
  method, 179
  of communication protocols, 182
  of hardware, 182
  of software, 182
  of systems, 265
  post-development, 180, 266
  pre-development, 180, 266
  process, 281
  program, 279
  property, 180
  property-oriented, 265
  semi-automatic, 265
  techniques, 179

weakest precondition, 286
while-statement, 270, 271
  body, 283, 293, 296
  non-termination, 303
wise-men puzzle, 356
Wolper, P., 263
word
  empty, 130
world
  accessible, 321
  possible, 321, 349

year-2000 problem, 267