COMP0009 Logic

Lecture notes · 2026–27

Part 2: Automated reasoning

Chapter 7

SAT Solving

Alexander V. Gheorghiu

Department of Computer Science, University College London

This chapter is issued on its own from the complete lecture notes for COMP0009 Logic. Its section, theorem, exercise and equation numbers are those of the complete notes, so that the two can be read side by side; references to other chapters and parts point into the complete notes.

Contents

7. SAT Solving 1
7.1. Satisfiability 2
7.2. Growing a Valuation 4
7.3. Conjunctive Normal Form 7
7.4. DPLL 12
7.5. DPLL in Practice 17
7.6. From Sentences to Clauses 22
7.7. Herbrand's Theorem 24
7.8. A Procedure for First-Order Logic 31
7.9. Prenex and Skolem Form *(optional)* 35
7.10. Resolution and Unification *(optional)* 42
History 54
Exercises 59

7. SAT Solving

The completeness theorem for NK says that whenever a sentence follows from others, it has a derivation from them. It does not say how to find one, and the search is hard to steer: at every step one may appeal to excluded middle for any formula at all, or instantiate a quantifier at any term, and nothing on the page says which choice leads anywhere.

Semantics offers a second route. By definition, $\Gamma \models \varphi$ fails exactly when some model of Γ makes φ false. So instead of looking for a derivation we may look for such a model, a counter-model. If the search finds one, the entailment fails, and the model shows it. If the search is systematic and fails, the entailment holds. This chapter is about making that search precise, mechanical and fast.

The search comes down to something simple. As far as a given question is concerned, a structure is pinned down by which basic facts hold in it: whether this element has that property, whether those two are related. Each basic fact is a switch, on or off, and a model is a setting of the switches that makes every sentence true. Programs that look for such settings are called *SAT solvers*.

SAT solvers are among the most used tools of logic. Whenever a problem comes down to "find a choice of yes-or-no settings that obeys all these constraints, or show that there is none", a solver can be put to work on it, and many problems do.

- *Hardware.* Chip makers check that an optimised circuit computes the same function as its specification, and search for inputs that drive a design into a bad state within a given number of clock cycles. A bug found this way costs nothing; one found after manufacture can cost a fortune.
- *Software.* Questions about programs (can this index go out of bounds? can these two versions of a function ever disagree? can anyone outside this account read this file?) are translated into satisfiability questions, usually through the SMT solvers of Remark 7.48, which run a SAT solver at their core.
- *Configuration.* Installing software means choosing versions of packages so that every dependency is met and no two conflict, and several package managers hand that problem to a SAT solver. Configuring a product with options that exclude or require one another is the same problem.
- *Planning and scheduling.* Timetables, rosters and plans are written as "at each step, which actions happen", and the solver finds a plan or shows that none exists.
- *Mathematics.* Solvers have settled open combinatorial questions by exhaustive search, among them the Boolean Pythagorean triples problem in 2016, with a certificate of some two hundred terabytes that an independent program checked.

In none of these does anybody write the solver's input by hand. The problem is stated in a natural language of its own, with quantifiers and arrows and arithmetic, and a program translates it into the shape the solver wants. This chapter studies both halves: the translation and the search.

The chapter first solves a simpler problem, and then reduces the real one to it.

- *Propositional.* Given constraints on finitely many switches, is there a setting that obeys them all?
  - Section 7.1 shows that entailment and validity are questions of satisfiability, and that once the things to be talked about are named, the question is propositional.
  - Section 7.2 grows a setting one switch at a time, and asks what the search must be able to read off its constraints.
  - Section 7.3 puts every constraint into the shape that makes this easy, conjunctive normal form, and draws it as a matrix through which a satisfying setting is a path.
  - Section 7.4 proves that the resulting search, DPLL, is correct, and Section 7.5 shows how a program runs it with sets and a single operation on them.
- *First-order.* Sections 7.6 to 7.8 name every "someone", prove that a search may look only at the things it has named, and hand the resulting switches to DPLL, level by level.

Two optional sections follow. Section 7.9 opens the black box of Section 7.6, and Section 7.10 shows that a failed search is itself a proof, that some such proofs must be long, and how the instances that can clash are found without listing the rest. The main line quotes some of their results but none of their proofs.

7.1. **Satisfiability.** This section makes the idea of the introduction exact. It names the property a solver decides, satisfiability, shows that entailment and validity reduce to it, and explains why the heart of the matter is propositional. Throughout, Σ is a signature, and structures and sentences are Σ-structures and Σ-sentences, as in Chapter 6.

Definition 7.1 (satisfiable)**.** A set of sentences is satisfiable if it has a model.

Formally, a set Γ of sentences is *satisfiable* if

$$\mathfrak{M} \models \Gamma \qquad \text{for some structure } \mathfrak{M},$$

and *unsatisfiable* otherwise; a sentence φ is satisfiable if $\{\varphi\}$ is.

Definition 7.2 (counter-model)**.** A counter-model is a structure that shows an entailment to fail.

Formally, a *counter-model* to $\Gamma \models \varphi$ is a structure $\mathfrak{M}$ with

$$\mathfrak{M} \models \Gamma \qquad \text{and} \qquad \mathfrak{M} \not\models \varphi.$$

Lemma 7.3 (reduction to satisfiability)**.** *For every set Γ of sentences and every sentence φ,*

(i) $\Gamma \models \varphi$ if and only if $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable;

(ii) φ is valid if and only if $\neg\varphi$ is unsatisfiable.

Proof. A model of $\Gamma \cup \{\neg\varphi\}$ is exactly a counter-model to $\Gamma \models \varphi$, and an entailment holds exactly when it has none. Part (ii) is part (i) with Γ empty. □

In words: every question about what follows from what, and about what is valid, is a question about whether some set of sentences has a model.

The lemma is close to a tautology, but it changes what we look for. A proof that $\Gamma \models \varphi$ may now be a failed attempt to build a model of $\Gamma \cup \{\neg\varphi\}$, provided the attempt is systematic enough to show that every way of building one collapses.

Example 7.4 (the company: a policy change and a bug report)**.** Harriet Ashworth looks after the platform of the company of Chapter 6. She has been asked to sign off two rules for its access policy, that engineers report to managers and that managers are not engineers:

$$\mathrm{P}_1 : \ \forall x\,(\mathsf{Eng}(x) \to \mathsf{Mgr}(\mathsf{boss}(x))),$$
$$\mathrm{P}_2 : \ \forall x\,(\mathsf{Mgr}(x) \to \neg\mathsf{Eng}(x)).$$

Their point is that no engineer reports to an engineer:

$$\mathrm{C} : \ \forall x\,(\mathsf{Eng}(x) \to \neg\mathsf{Eng}(\mathsf{boss}(x))).$$

Harriet looks for a counter-model. Suppose some engineer, call her a, reports to an engineer. By P_1, a's boss is a manager; by P_2, a's boss is not an engineer; but she is. Every line was forced by the lines above it, so there is no counter-model, and $\{\mathrm{P}_1, \mathrm{P}_2\} \models \mathrm{C}$.

A colleague proposes dropping P_2, so that a senior engineer can lead a team. The same search now runs out of demands instead of collapsing. Suppose an engineer a reports to an engineer b. By P_1, b is a manager, and nothing forbids that; by P_1 again, b's boss is a manager, and the simplest choice is b herself. What is left is a structure: two engineers, with b a manager and the boss of both, ceo naming b, Richer empty and $=$ identity. It satisfies P_1 and falsifies C at a. That is not a failed proof but a bug report: the weakened policy allows an engineer who manages an engineer. (Example 7.77 finds a counter-model with one person.)

So the question to hand to a program is whether a finite set of sentences has a model. We call the members of such a set its *constraints*, since each is a sentence the model must make true, and a program that decides whether a finite set of constraints has a model is a *solver*. We ask more of a solver than a verdict: we want evidence that can be checked without trusting it. For "yes" the evidence is a model, which, if finite, anyone can check by evaluating the sentences in it. For "no" it must be a finite object showing that every attempt to build a model collapses; Section 7.4 builds one.

Switches. A solver cannot list structures: a model may have five elements, or be the natural numbers, and there is no list to go through. It does what the search of Example 7.4 did. It names the things it needs, and records for each atomic sentence about them, such as $\mathsf{Eng}(a)$ or $\mathsf{Mgr}(\mathsf{boss}(a))$, whether it holds. Each atomic sentence is a switch, and what the search builds is a setting of the switches. A setting can be written down, finitely many switches have finitely many settings, and a search through them can be seen

to have failed. Section 7.7 shows that nothing is lost: a set of constraints has a model exactly when some setting of the switches about its names obeys every instance of the constraints. The core of the problem is therefore propositional, and the next four sections solve it.

Propositional logic. Once the switches are fixed, what is inside an atomic sentence plays no part, and $\mathsf{Eng}(\mathsf{boss}(a))$ might as well be a single letter. So until Section 7.6 we fix a set $\mathbb{P}$ of *letters*, the switches of the search, write p, q, r for letters, and build formulas by the grammar

$$\varphi, \psi \ ::= \ p \ \mid \ \bot \ \mid \ \neg\varphi \ \mid \ (\varphi \to \psi) \ \mid \ (\varphi \wedge \psi) \ \mid \ (\varphi \vee \psi),$$

with the bracket conventions of Chapter 6 and $\top$ abbreviating $\neg\bot$. Here $\neg$ is a connective in its own right rather than an abbreviation of $\varphi \to \bot$. The two readings always agree in truth value, and having $\neg$ as a connective lets the translations of Section 7.3 say directly what they do to a negation.

A structure for a signature of letters amounts to a *valuation*

$$v : \mathbb{P} \to \{\text{true}, \text{false}\}$$

(the propositional case of satisfaction), which is exactly a setting of the switches. We write $v \models \varphi$, $v \models \Gamma$ and "satisfiable" with the meanings of Chapter 6. One more notion is needed: when two formulas allow the same settings, one may replace the other.

Definition 7.5 (equivalent formulas)**.** Two formulas are equivalent when every valuation gives them the same value.

Formally, $\varphi \equiv \psi$ if, for every valuation v,

$$v \models \varphi \qquad \text{iff} \qquad v \models \psi.$$

Equivalence is a congruence: replacing a part of a formula by an equivalent part gives an equivalent formula, since the value of a formula is computed from the values of its parts.

7.2. **Growing a Valuation.** A search for a satisfying valuation could try the rows of the truth table one by one. That is correct and hopeless: a hundred letters give more rows than there are stars in the observable universe, and the problems solvers meet in practice have millions of letters. A search by hand does better. It sets one switch at a time, and stops as soon as the constraints are settled one way or the other. This section sets up the language for such a search: what a valuation looks like half-built, how it grows, and the one move that is always safe. It ends with the question that shapes the rest of the propositional part: how can the search tell that a half-built valuation has settled a constraint?

Partial valuations. Halfway through the search some letters have a value and the rest have none yet. Two definitions describe that state: a literal says how one switch is set, and a partial valuation says how several are.

Definition 7.6 (literal)**.** A literal says that one switch is on, or that it is off.

Formally, a *literal* is a letter p or a negated letter $\neg p$. We write ℓ for a literal, and $\bar{\ell}$ for its *complement*:

$$\bar{p} \coloneqq \neg p \qquad \text{and} \qquad \overline{\neg p} \coloneqq p.$$

Definition 7.7 (partial valuation)**.** A partial valuation gives some letters a truth value and leaves the rest undecided.

Formally, a *partial valuation* is a partial function

$$u : \mathbb{P} \rightharpoonup \{\text{true}, \text{false}\}$$

defined on finitely many letters. A letter p is *decided* by u if $u(p)$ is defined, and *undecided* otherwise. On literals we set

$$u(\neg p) \coloneqq \textbf{not}\, u(p) \quad \text{for } p \text{ decided by } u, \qquad \text{and} \qquad u \models \ell \quad \text{iff} \quad u(\ell) = \text{true}.$$

The partial valuation that decides nothing is written $\varnothing$.

A partial valuation is not a valuation with gaps filled in arbitrarily: on an undecided letter p it has no opinion, and neither $u \models p$ nor $u \models \neg p$ holds. Writing $\varnothing$ for the one that decides nothing is apt, since a function defined nowhere is, as a set of pairs, empty.

Definition 7.8 (extension)**.** One partial valuation extends another when it keeps everything the other has decided.

Formally, a partial valuation or valuation w *extends* a partial valuation u, written $u \preceq w$, if

$$w(p) = u(p) \qquad \text{for every letter } p \text{ decided by } u.$$

Three facts follow at once.

- Extension is transitive: if $u \preceq w$ and $w \preceq v$, then $u \preceq v$.
- Everything extends $\varnothing$.
- What u makes true, every extension makes true: if $u \preceq w$ and $u \models \ell$, then $w \models \ell$.

Definition 7.9 (adding a literal)**.** Adding a literal sets one more switch, so as to make the literal true.

Formally, for a partial valuation u and a literal ℓ whose letter p is undecided by u, the partial valuation $u + \ell$ agrees with u on the letters u decides, decides nothing else besides p, and sets

$$(u + p)(p) \coloneqq \text{true} \qquad \text{and} \qquad (u + \neg p)(p) \coloneqq \text{false}.$$

So $u + p$ is "u with p on" and $u + \neg p$ is "u with p off"; unlike the update $s[d/x]$ of an assignment, it only fills a blank. A partial valuation is fixed by the literals it makes true, so we name it by listing them in braces: $\{p, \neg q\}$ is the partial valuation with p on, q off and every other letter undecided, the one reached from $\varnothing$ by adding p and then $\neg q$. In this notation

$$u + \ell = u \cup \{\ell\},$$

and $\varnothing$, the partial valuation that decides nothing, is the empty list, as its name says. The proofs below use one fact about adding a literal, which is read off the definition: for every valuation v,

$$u + \ell \preceq v \qquad \text{iff} \qquad u \preceq v \text{ and } v \models \ell.$$

Definition 7.10 (alive, dead)**.** A partial valuation is alive for a set of formulas if some way of finishing it satisfies the set, and dead otherwise.

Formally, u is *alive* for Γ if

$$v \models \Gamma \qquad \text{for some valuation } v \text{ with } u \preceq v,$$

and *dead* for Γ otherwise.

In words: u is alive when a search that has made the choices recorded in u can still succeed. So a set is satisfiable exactly when $\varnothing$ is alive for it, and the search decides this by following the question down, from u to $u + \ell$. For a single formula φ, "alive for φ" means alive for $\{\varphi\}$.

Splitting. The first move needs no information at all: pick an undecided letter and try both values.

Lemma 7.11 (splitting)**.** *Let Γ be a set of formulas, u a partial valuation and p a letter undecided by u. Then*

$$u \text{ is alive for } \Gamma \qquad \text{iff} \qquad u + p \text{ or } u + \neg p \text{ is alive for } \Gamma.$$

Proof. A valuation that extends $u + p$ or $u + \neg p$ extends u. Conversely, a valuation that extends u makes p true or false, and so extends $u + p$ or $u + \neg p$. □

The choice of a letter and the two tries are a *decision*. The lemma says that a decision loses nothing, provided both sides are tried.

Remark 7.12 (why the search must be able to go back). It is tempting to try one side only: go through the letters in order and set each for good, on unless that makes some constraint false. On the constraints

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

in the order p, q, r, this sets p on (no constraint is yet false), and then either value of q falsifies a constraint, and the method is stuck. Yet p off and r on satisfy all three. Without the other side of the split, a search can answer "no" when the answer is "yes".

What the search must read off. Splitting is always safe, but on its own it is the truth table again, one letter at a time. What makes a search fast is knowing when to stop and when not to guess. Under a partial valuation u the search wants to know, for each constraint, which of three things holds:

- every valuation that extends u satisfies it, so it can be forgotten;
- no valuation that extends u satisfies it, so u is dead;
- it has only one way left to be satisfied, so the next switch is forced.

For an arbitrary formula none of these can be seen by looking. Whether $\varnothing$ is dead for $\{\varphi\}$ is the question whether φ is satisfiable, which is the very problem we are trying to solve.

For one shape of constraint, though, all three are visible at a glance. A disjunction of switches set on or off, such as

$$\neg p \vee \neg q \vee r,$$

is satisfied by every extension of u as soon as u makes one disjunct true; it is dead as soon as u makes every disjunct false; and when u makes all disjuncts but one false, the last one must be made true. A conjunction of such disjunctions is satisfied when each conjunct is, and dead when one conjunct is. So we want every constraint in that shape. The shape is called conjunctive normal form, and it is the subject of the next section.

7.3. **Conjunctive Normal Form.** Section 7.2 asked for constraints whose status under a partial valuation can be read at a glance, and found the shape: a conjunction of disjunctions of switches set on or off. It is also the shape every constraint has in principle. A constraint can only forbid: its value is fixed by the values of its letters, so all it can do is rule out some combinations of them. The constraint $p \wedge q \to r$ is false in exactly one row of its truth table, the one with p on, q on and r off, so it says no more than "not that row", that is, "p off, or q off, or r on": the disjunction $\neg p \vee \neg q \vee r$. A constraint that forbids several rows is several such disjunctions, joined by $\wedge$. This section defines the shape, shows how to reach it from any formula in two stages, and draws it as a matrix in which a satisfying valuation is a path.

Definition 7.13 (clause)**.** A clause is a list of literals of which at least one must hold.

Formally, a *clause* is a disjunction

$$\ell_1 \vee \cdots \vee \ell_n$$

of literals, with $n \geq 0$; its *literals* are $\ell_1, \ldots, \ell_n$. For $n = 0$ it is $\bot$, the *empty clause*, written $\square$. A clause with one literal is a *unit clause*. We write C, D for clauses.

A valuation satisfies a clause when it makes one of its literals true; the empty clause is satisfied by nothing. Since $\vee$ is commutative, associative and idempotent, neither the order of the literals nor repetition changes what a clause says: $p \vee q$, $q \vee p$ and $p \vee q \vee p$ are equivalent.

Definition 7.14 (conjunctive normal form)**.** A formula is in conjunctive normal form when it is a list of clauses, all of which must hold.

Formally, a formula is in *conjunctive normal form* if it is a conjunction

$$C_1 \wedge \cdots \wedge C_m$$

of clauses, with $m \geq 0$; for $m = 0$ it is $\top$. Its *clauses* are $C_1, \ldots, C_m$.

A valuation satisfies a formula in conjunctive normal form exactly when it satisfies each of its clauses. A finite set of formulas in conjunctive normal form can be read as the single conjunction of all their clauses, and we do so without comment.

Example 7.15 (the platform: five rules)**.** Harriet's platform has a hundred feature switches and a file of rules that they must obey together. Three of the switches are `fast-sync`, `offline-mode` and `compression`, written p, q and r, and five rules concern them: `fast-sync` needs `offline-mode`; `offline-mode` needs `fast-sync`; the mobile app needs at least one of the two; with both on, `compression` must be on, or the sync saturates the link; and with both on, `compression` must be off, since the offline cache cannot

read compressed packets. As clauses, with the pattern each forbids:

$$\begin{aligned}
C_1 &= \neg p \vee q && \text{never } p \text{ on, } q \text{ off;}\\
C_2 &= \neg q \vee p && \text{never } q \text{ on, } p \text{ off;}\\
C_3 &= p \vee q && \text{never } p \text{ off, } q \text{ off;}\\
C_4 &= \neg p \vee \neg q \vee r && \text{never } p \text{ on, } q \text{ on, } r \text{ off;}\\
C_5 &= \neg p \vee \neg q \vee \neg r && \text{never } p \text{ on, } q \text{ on, } r \text{ on.}
\end{aligned}$$

The rule file is the formula $C_1 \wedge \cdots \wedge C_5$ in conjunctive normal form. No setting avoids all five patterns: the first three force p and q on, and then C_4 and C_5 disagree about r. The file has a bug; Example 7.25 shows it in the matrix, and Example 7.34 shows DPLL finding it.

Two stages. Reading the clauses off the truth table is out of the question, since thirty letters give a billion rows. The conversion works on the formula itself, in two stages.

1. Push every negation down to the letters and remove the arrows, so that only $\wedge$ and $\vee$ stand above the literals. After this stage every letter occurs as a switch that is plainly on or plainly off.
2. Push every $\vee$ below every $\wedge$. After this stage the formula is a list of clauses.

Stage 1: negation normal form. The negations are pushed down by De Morgan's laws, double negation and the truth table of the arrow,[1] each checked by a truth table of four rows or two:

$$\neg(\psi \wedge \chi) \equiv \neg\psi \vee \neg\chi, \qquad \neg(\psi \vee \chi) \equiv \neg\psi \wedge \neg\chi, \qquad \neg\neg\psi \equiv \psi,$$
$$\psi \to \chi \equiv \neg\psi \vee \chi, \qquad \neg(\psi \to \chi) \equiv \psi \wedge \neg\chi.$$

Definition 7.16 (negation normal form)**.** A formula is in negation normal form when negation occurs only on letters and the only other connectives are $\wedge$ and $\vee$.

Formally, a formula is in *negation normal form* if it is built from literals, $\bot$ and $\top$ by $\wedge$ and $\vee$ alone.

Definition 7.17 (the translation into negation normal form)**.** The translation applies the laws above from the outside in.

[1] The arrow's truth table makes a conditional true whenever its antecedent is false, so that "if the moon is made of cheese, then $2 + 2 = 4$" comes out true. The table is Philo of Megara's, from the fourth century BC, and the argument about it is as old: by Sextus Empiricus's report, the poet Callimachus wrote that even the crows on the rooftops were cawing about which conditionals are true. For how to live with it, seek spiritual guidance from your nearest philosopher, or from the History of Chapter 6.

Formally, $\mathrm{nnf}(\varphi)$ is defined by recursion on the size of φ, by the equations

$$\begin{aligned}
\mathrm{nnf}(\varphi) &\coloneqq \varphi \qquad \text{for } \varphi \text{ one of } p,\ \neg p,\ \bot,\ \neg\bot,\\
\mathrm{nnf}(\psi \wedge \chi) &\coloneqq \mathrm{nnf}(\psi) \wedge \mathrm{nnf}(\chi),\\
\mathrm{nnf}(\psi \vee \chi) &\coloneqq \mathrm{nnf}(\psi) \vee \mathrm{nnf}(\chi),\\
\mathrm{nnf}(\psi \to \chi) &\coloneqq \mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\chi),\\
\mathrm{nnf}(\neg\neg\psi) &\coloneqq \mathrm{nnf}(\psi),\\
\mathrm{nnf}(\neg(\psi \wedge \chi)) &\coloneqq \mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\neg\chi),\\
\mathrm{nnf}(\neg(\psi \vee \chi)) &\coloneqq \mathrm{nnf}(\neg\psi) \wedge \mathrm{nnf}(\neg\chi),\\
\mathrm{nnf}(\neg(\psi \to \chi)) &\coloneqq \mathrm{nnf}(\psi) \wedge \mathrm{nnf}(\neg\chi).
\end{aligned}$$

Exactly one equation applies to each formula. The recursion is on size rather than shape, since an equation may call nnf on $\neg\psi$, which is not a part of φ; but $\neg\psi$ is smaller than φ, so the recursion ends.

Lemma 7.18 (negation normal form is faithful)**.** *For every formula φ, the formula* $\mathrm{nnf}(\varphi)$ *is in negation normal form, and* $\mathrm{nnf}(\varphi) \equiv \varphi$.

Proof. The shape of $\mathrm{nnf}(\varphi)$ is read off the equations, each of which returns a literal, $\bot$ or $\top$, or combines results of nnf by $\wedge$ or $\vee$. The equivalence is by induction on the size of φ. The first equation is trivial; for $\wedge$ and $\vee$ use the induction hypothesis and the congruence; each of the other five replaces φ by the right-hand side of one of the laws and applies the induction hypothesis to the smaller formulas there, as in

$$\mathrm{nnf}(\neg\psi) \vee \mathrm{nnf}(\neg\chi) \ \equiv\ \neg\psi \vee \neg\chi \ \equiv\ \neg(\psi \wedge \chi).$$

□

Stage 2: multiplying out. The law that moves $\vee$ below $\wedge$ is distributivity,

$$\psi \vee (\chi_1 \wedge \chi_2) \ \equiv\ (\psi \vee \chi_1) \wedge (\psi \vee \chi_2),$$

school algebra's $a(b + c) = ab + ac$ with $\vee$ for times and $\wedge$ for plus. Two conjunctions of clauses multiply out like $(a + b)(c + d)$, every clause of one against every clause of the other:

$$(C_1 \wedge C_2) \vee (D_1 \wedge D_2) \ \equiv\ (C_1 \vee D_1) \wedge (C_1 \vee D_2) \wedge (C_2 \vee D_1) \wedge (C_2 \vee D_2).$$

For clauses C and D we write $C \vee D$ for the clause whose literals are those of C followed by those of D.

Lemma 7.19 (multiplying out)**.** *For formulas $C_1 \wedge \cdots \wedge C_m$ and $D_1 \wedge \cdots \wedge D_k$ in conjunctive normal form,*

$$(C_1 \wedge \cdots \wedge C_m) \vee (D_1 \wedge \cdots \wedge D_k) \ \equiv \bigwedge_{i \le m,\ j \le k} (C_i \vee D_j).$$

Proof. Under a valuation v the right-hand side is false exactly when some $C_i \vee D_j$ is, that is, when some C_i and some D_j are both false, which is exactly when both conjunctions on the left are false. □

In words: a disjunction of two lists of demands is the list of all pairwise disjunctions of their demands, just as $(a + b)(c + d)$ is $ac + ad + bc + bd$.

Definition 7.20 (the translation into conjunctive normal form)**.** The translation multiplies out every $\vee$ over every $\wedge$, from the inside out.

Formally, for ν in negation normal form, $\mathrm{cnf}(\nu)$ is defined by recursion on the shape of ν, by the equations

$$\begin{aligned}
\mathrm{cnf}(\ell) &\coloneqq \ell \qquad \text{for a literal } \ell,\\
\mathrm{cnf}(\bot) &\coloneqq \square,\\
\mathrm{cnf}(\top) &\coloneqq \top,\\
\mathrm{cnf}(\nu_1 \wedge \nu_2) &\coloneqq \mathrm{cnf}(\nu_1) \wedge \mathrm{cnf}(\nu_2),\\
\mathrm{cnf}(\nu_1 \vee \nu_2) &\coloneqq \bigwedge_{i\le m,\ j\le k} (C_i \vee D_j),
\end{aligned}$$

where in the last equation $\mathrm{cnf}(\nu_1) = C_1 \wedge \cdots \wedge C_m$ and $\mathrm{cnf}(\nu_2) = D_1 \wedge \cdots \wedge D_k$, and in the one before it the conjunction lists the clauses of both sides. The *conjunctive normal form* of an arbitrary formula φ is $\mathrm{cnf}(\mathrm{nnf}(\varphi))$.

Proposition 7.21 (the conjunctive normal form is equivalent)**.** *For every formula φ, the formula* $\mathrm{cnf}(\mathrm{nnf}(\varphi))$ *is in conjunctive normal form, and* $\mathrm{cnf}(\mathrm{nnf}(\varphi)) \equiv \varphi$*.*

Proof. By the faithfulness of negation normal form (Lemma 7.18) it suffices that $\mathrm{cnf}(\nu) \equiv \nu$ for ν in negation normal form, by induction on the shape of ν: the first three equations change nothing, the fourth is the induction hypothesis with associativity of $\wedge$, and the fifth is the induction hypothesis with the lemma on multiplying out (Lemma 7.19). The shape of the result is read off the equations. $\square$

Example 7.22 (a conjunctive normal form)**.** Take $\varphi = \neg((p \to q) \wedge \neg r)$. Stage 1 gives

$$\begin{aligned}
\mathrm{nnf}(\varphi) &= \mathrm{nnf}(\neg(p \to q)) \vee \mathrm{nnf}(\neg\neg r) && \text{the equation for } \neg(\psi \wedge \chi),\\
&= (p \wedge \neg q) \vee r && \text{the equations for } \neg(\psi \to \chi),\ \neg\neg\psi,
\end{aligned}$$

and Stage 2 multiplies out:

$$\mathrm{cnf}\big((p \wedge \neg q) \vee r\big) \;=\; (p \vee r) \wedge (\neg q \vee r).$$

The two clauses forbid exactly the rows that make φ false: p and r off, and q on with r off.

The matrix. A formula in conjunctive normal form is a switchboard, and it reads best laid out as one. Write it as a *matrix*, with one column for each clause and the literals of the clause stacked in its column. For instance,

$$(\neg p \vee q) \wedge (p \vee q) \wedge (\neg q \vee r) \qquad \text{is written} \qquad \begin{bmatrix} \neg p & p & \neg q \\ q & q & r \end{bmatrix}.$$

A valuation that satisfies the formula makes some entry of every column true. Reading one such entry from each column, left to right, traces a way through the matrix, and the entries on it are never complementary, since a valuation cannot make both p and $\neg p$ true. That is the whole of satisfiability.

Definition 7.23 (path)**.** A path through a formula in conjunctive normal form picks one literal from each clause.

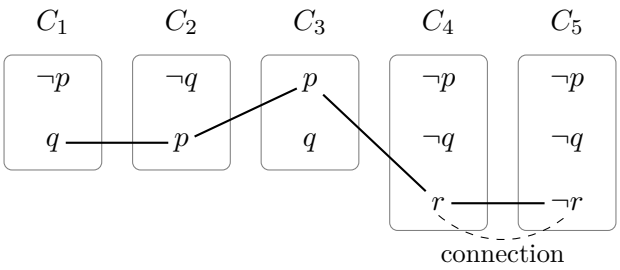


FIGURE 7.1. The matrix of the five rules, with one path.

Formally, let $\varphi = C_1 \wedge \cdots \wedge C_m$ be in conjunctive normal form. A *path* through φ is a sequence $(\ell_1, \ldots, \ell_m)$ in which each ℓ_i is a literal of C_i. Two complementary literals on a path form a *connection*, and a path without a connection is *consistent*.

In the matrix above, (q, q, r) is a consistent path and $(\neg p, p, \neg q)$ is not: its first two entries form a connection. A formula with the empty clause among its clauses has no paths at all, since nothing can be picked from an empty column; the formula $\top$, with no clauses, has exactly one path, the empty one, and it is consistent.

Lemma 7.24 (paths)**.** *A valuation satisfies a formula in conjunctive normal form if and only if it makes every literal of some path through it true. Hence the formula is satisfiable if and only if some path through it is consistent.*

Proof. Let $\varphi = C_1 \wedge \cdots \wedge C_m$. If $v \models \varphi$, choose for each i a literal of C_i that v makes true; the choices form a path whose literals v makes true. Conversely, a valuation that makes every literal of a path true makes a literal of each clause true. For the second claim, a path whose literals are all true under one valuation has no connection; and a consistent path has a valuation making all its literals true, namely one that sets each letter on the path so as to make its occurrences there true, which consistency allows, and sets the other letters at will. □

In words: a formula in conjunctive normal form is satisfiable exactly when there is a way through its matrix that never throws a switch both ways. Section 7.4 searches for such a way without listing the paths, whose number is the product of the lengths of the clauses.

Example 7.25 (the platform, cont'd: no way through)**.** The matrix of the five rules of Example 7.15 is drawn in Figure 7.1, with one of its $2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 = 72$ paths. Every path through C_1, C_2, C_3 without a connection contains both p and q; a path that continues without a connection must then take r from C_4 and $\neg r$ from C_5, and those two form a connection. So no path is consistent, and by the lemma no setting obeys the five rules.

Remark 7.26 (looking for connections). The matrix and its paths are the picture behind the *connection method* of automated reasoning, which looks at the lemma on paths from the other side. Instead of searching for one

consistent path, it shows that there is none: a formula is unsatisfiable exactly when every path through its matrix contains a connection, so a refutation can be presented as a set of connections that between them block every path.[2] The search of Section 7.4 can be read the same way: a closed branch shows that every path through the matrix clashes with the literals set on that branch.

The cost, and a way round it. Multiplying out multiplies the counts. The formula

$$(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee \cdots \vee (p_n \wedge q_n)$$

has $2n$ letters and 2^n clauses in its conjunctive normal form, one for each way of choosing p_i or q_i from every pair; thirty pairs give a billion clauses. Solvers avoid this by giving up equivalence and keeping only what they need, the verdict.

Definition 7.27 (equisatisfiable)**.** Two sets of formulas are equisatisfiable when both are satisfiable or neither is.

Formally, sets Γ and Δ of formulas, propositional or first-order, are *equisatisfiable* if

$$\Gamma \text{ is satisfiable} \qquad \text{iff} \qquad \Delta \text{ is satisfiable.}$$

Equisatisfiability keeps the verdict but not the evidence: $\{p\}$ and $\{q\}$ are equisatisfiable, and a valuation that satisfies one need not satisfy the other. So every rewriting of this chapter that is not an equivalence comes with a *way back*: a way of turning whatever satisfies the new set into something that satisfies the old one.

Remark 7.28 (at work: Tseitin's translation). Each compound subformula ψ of the input gets a new letter a_ψ, with at most three clauses saying what it means. For $\psi = \chi_1 \wedge \chi_2$, with letters b and c for the parts, the clauses $\neg a_\psi \vee b$, $\neg a_\psi \vee c$ and $a_\psi \vee \neg b \vee \neg c$ say that a_ψ is on exactly when b and c are; the other connectives are alike, and one unit clause a_φ asserts the whole formula. For n pairs that is $3(2n-1)+1$ clauses rather than 2^n. The result is not equivalent to φ, since it mentions new letters, but it is equisatisfiable with $\{\varphi\}$, and the way back is to forget the new letters (Exercise 7.3).[3]

From here on, a constraint without quantifiers is a formula in conjunctive normal form.

7.4. **DPLL.** The search of Section 7.2 can now be run on constraints in the shape of Section 7.3. Let Γ be a finite set of formulas, ψ a formula, and φ the conjunction of the clauses of the conjunctive normal forms of the members of $\Gamma \cup \{\neg\psi\}$. Then

$\Gamma \models \psi$	iff	$\Gamma \cup \{\neg\psi\}$ is unsatisfiable	(Lemma 7.3)
	iff	φ is unsatisfiable	(Proposition 7.21)
	iff	every path through φ has a connection	(Lemma 7.24).

[2]The method is Bibel's, and Andrews's *matings* are the same idea; both were developed for first-order logic around 1980, where a connection may need a substitution to become one.

[3]The translation is Tseitin's (1968); the History says more.

The three steps are the reduction lemma, the equivalence of the normal form, and the lemma on paths. So the task is to find a valuation that satisfies a formula in conjunctive normal form, or a consistent path through its matrix, or to show that there is none. A valuation that is found is a counter-model in miniature, a setting under which Γ holds and ψ fails; a proof that there is none is a proof that $\Gamma \models \psi$. This section makes the three readings of Section 7.2 precise for clauses, adds the second move of the search, the forced move, and proves that the resulting search, DPLL, is correct.

Reading the clauses. Section 7.2 asked for three readings of a constraint under a partial valuation, and for a clause each is a matter of looking at its literals.

Definition 7.29 (the status of a clause)**.** Under a partial valuation a clause may be satisfied, falsified, or a unit.

Formally, let u be a partial valuation. A clause C is

- *satisfied* under u, written $u \models C$, if $u \models \ell$ for some literal ℓ of C;
- *falsified* under u if $u \models \bar{\ell}$ for every literal ℓ of C;
- a *unit* under u, with *undecided literal* ℓ, if ℓ is a literal of C whose letter u does not decide, and $u \models \bar{\ell'}$ for every literal ℓ' of C other than ℓ.

We write $u \models \varphi$ if $u \models C$ for every clause C of φ.

For instance, under $u = \{p, \neg q\}$:

$p \vee s$	is satisfied, since $u \models p$;
$\neg p \vee q$	is falsified, since $u \models p$ and $u \models \neg q$;
$q \vee r$	is a unit with undecided literal r, since $u \models \neg q$;
$\neg p \vee r \vee s$	is none of these: two of its letters are undecided.

The empty clause is falsified under every partial valuation, since it has no literal to make true.

In the matrix, a partial valuation ticks every entry it makes true and crosses out every entry it makes false. A column with a tick is satisfied; a column whose entries are all crossed out is falsified; and a column with no tick and a single entry left is a unit, and every consistent path that agrees with u must go through that entry.

Lemma 7.30 (the status of a clause)**.** *Let u be a partial valuation, C a clause and v a valuation with $u \preceq v$.*

(i) If $u \models C$, then $v \models C$.

(ii) If C is falsified under u, then $v \not\models C$.

(iii) If C is a unit under u with undecided literal ℓ, then $v \models C$ if and only if $v \models \ell$.

Proof. Every literal true under u is true under v. In (i) some literal of C is true under u; in (ii) every literal of C is false under u, hence under v; in (iii) every literal other than ℓ is false under v, so C is true exactly when ℓ is. □

In words: however the blanks are filled in, a satisfied clause stays satisfied and a falsified clause stays falsified, and a unit says how one blank must be

filled. In particular, a partial valuation under which some clause of φ is falsified is dead for φ, and one with $u \models \varphi$ is alive.

Propagation. The second move of the search, after splitting, uses the third reading.

Lemma 7.31 (propagation: the other branch is empty)**.** *Let φ be a formula in conjunctive normal form and u a partial valuation. If a clause C of φ is a unit under u with undecided literal ℓ, then $u + \bar{\ell}$ is dead for φ, and so*

$$u \text{ is alive for } \varphi \qquad \text{iff} \qquad u + \ell \text{ is alive for } \varphi.$$

Proof. A valuation that extends $u + \bar{\ell}$ extends u and makes ℓ false, so it falsifies C by part (iii) of the lemma on the status of a clause (Lemma 7.30). The rest is the splitting lemma (Lemma 7.11). □

In words: when a clause has only one way left to be satisfied, the search may take that way without losing anything. Its literal is *forced*: a decision on its letter would be a waste, since one side of it is dead before it is tried. *Unit propagation* makes forced moves for as long as some clause is a unit and none is falsified. Each forced move may create new units, and on a good day propagation alone ends the search.

Chains of extensions. With both moves in hand, a search builds a chain of partial valuations, each the one before with a single literal added:

$$\varnothing \preceq \{\ell_1\} \preceq \{\ell_1, \ell_2\} \preceq \cdots \preceq \{\ell_1, \ldots, \ell_k\}.$$

Each literal is forced by a unit or is one side of a decision, and by transitivity a valuation that extends the last link extends every link. Call the last link u. The chain stops in one of two ways.

- $u \models \varphi$. Every valuation that extends u satisfies φ, and the search has found what it wanted.
- *Some clause is falsified under u.* Then u is dead. The search backs up to the last decision and starts a new chain from its other side.

Nothing is lost on the way: a forced step keeps every satisfying valuation, by propagation (Lemma 7.31), and a decision keeps them all between its two sides, by splitting (Lemma 7.11), provided the other side is tried when this one dies.

DPLL trees. Decisions, forced moves, and backing up from a dead end to the other side of the last decision make a method called DPLL, after the initials of the four people who wrote it down, in two papers two years apart.[4] A run of it is a tree, grown from the root down: each node carries the literal a move added, and a decision forks. Figure 7.2 gives the rules: when what is above the bar holds of a branch, what is below may be written under it, and a vertical bar forks the branch.

Definition 7.32 (DPLL tree)**.** A DPLL tree is a tree of literals grown from the root by the rules of Figure 7.2.

[4] Davis and Putnam (1960) and Davis et al. (1962); the History says what each paper did.

$$\frac{C \text{ of } \varphi \text{ a unit on } B, \text{ undecided literal } \ell}{\ell}\ \text{unit} \qquad \frac{p \text{ a letter of } \varphi \text{ undecided on } B}{p \mid \neg p}\ \text{decide}$$

$$\frac{C \text{ of } \varphi \text{ falsified on } B}{\times}\ \text{close} \qquad \frac{u_B \models \varphi}{\checkmark}\ \text{finish}$$

FIGURE 7.2. The rules of DPLL.

Formally, a *DPLL tree for* a formula φ in conjunctive normal form is a finite tree whose root carries no literal and whose other nodes carry literals, obtained from the root alone by applying *unit* and *decide*, each at the foot of a branch on which no clause of φ is falsified. The *partial valuation* of a branch B whose literals, from the root down, are $\ell_1, \ldots, \ell_k$ is $u_B \coloneqq \{\ell_1, \ldots, \ell_k\}$, and a clause is satisfied, falsified or a unit *on* B if it is so under u_B.

In words: a DPLL tree is the record of a search. Each branch is one chain of extensions, and its literals are the decisions and forced moves made along it. Each rule adds a literal on a letter the branch leaves undecided, so u_B is defined. The rules *close* and *finish* add no literal; they mark the foot of a branch with $\times$ or $\checkmark$.

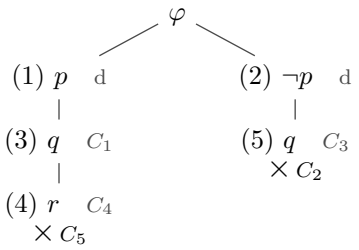
Definition 7.33 (closed and complete)**.** A branch is closed when it ends in a cross, finished when it ends in a tick, and complete when it ends in either.

Formally, a branch B of a DPLL tree for φ is

- *closed* if some clause of φ is falsified on B, and *open* otherwise;
- *finished* if $u_B \models \varphi$;
- *complete* if it is closed or finished.

The tree is *closed* if every branch is closed, and *complete* if every branch is complete.

Example 7.34 (the platform, cont'd: the switches that follow)**.** Harriet runs the search on the five rules of Example 7.15. No clause is a unit, so she decides p, on first. Beside each node is the rule that wrote it, d for a decision and a clause for a forced move; a cross carries the clause it falsifies.



Branch by branch:

under $\{p\}$,	C_1 forces q;
under $\{p, q\}$,	C_4 forces r;
under $\{p, q, r\}$,	C_5 is falsified;
under $\{\neg p\}$,	C_3 forces q;
under $\{\neg p, q\}$,	C_2 is falsified.

The tree is closed: five nodes and one decision, against eight rows of the truth table.

Remark 7.35 (what a closed branch knows). In Example 7.34 everything under node 1 closed, so $\{p\}$ is dead: the five rules entail $\neg p$, though none of them says so. In general, when every branch below a node closes, the partial valuation there is dead (the proof of soundness below shows it), and φ entails the clause made of the complements of the literals on the way down. Such a clause is worth keeping, since it records a dead end the search need never visit again; the optional Section 7.10 shows how solvers keep them, as *learned* clauses, and what the closed tree as a whole proves.

Correctness. Three things need proof: that a search can be carried to the end, that a closed tree is never wrong, and that an unsatisfiable formula always yields one.

Lemma 7.36 (termination)**.** *Let φ be a formula in conjunctive normal form with n letters.*

(i) No branch of a DPLL tree for φ has more than n nodes below the root.

(ii) There is a complete DPLL tree for φ.

Proof. Each rule adds a literal on an undecided letter, which gives (i). For (ii), apply rules to incomplete branches for as long as possible; by (i) this stops, after fewer than 2^{n+1} nodes. A branch that is neither closed nor finished has a clause neither satisfied nor falsified, hence an undecided letter, and *decide* applies; so when nothing applies the tree is complete. □

Theorem 7.37 (soundness of DPLL)**.** *If a formula φ in conjunctive normal form is satisfiable, then no DPLL tree for φ is closed.*

Proof. We prove the contrapositive by working up a closed tree from its leaves: the status of a clause kills the leaves, and splitting and propagation carry death up to the root. For a node N let u_N be the partial valuation of the branch from the root to N; we show by induction on the number of nodes below N that u_N is dead for φ.

- *N is a leaf.* Its branch is closed, so some clause is falsified under u_N, and u_N is dead.
- *N has two children, by a decision on p.* They carry $u_N + p$ and $u_N + \neg p$, both dead, so u_N is dead by splitting (Lemma 7.11).
- *N has one child, by unit with literal ℓ.* It carries $u_N + \ell$, which is dead, so u_N is dead by propagation (Lemma 7.31).

At the root, $\varnothing$ is dead, so no valuation satisfies φ. □

In words: a closed DPLL tree is a certificate of unsatisfiability. Exercise 7.6 gives a second proof, which follows a satisfying valuation down the tree instead.

Proposition 7.38 (finished branches)**.** *If B is a finished branch of a DPLL tree for φ, then $v \models \varphi$ for every valuation v with $u_B \preceq v$.*

Proof. Part (i) of the lemma on the status of a clause (Lemma 7.30), for each clause C of φ. □

Theorem 7.39 (completeness of DPLL)**.** *If a formula φ in conjunctive normal form is unsatisfiable, then every complete DPLL tree for φ is closed.*

Proof. An open branch of a complete tree is finished, and by the proposition every valuation extending it satisfies φ; there is one, setting the remaining letters false. □

In words: when φ is unsatisfiable, every search carried to the end produces the certificate, whatever its choices. Reading a valuation off an open branch is the truth lemma of the completeness proof of Chapter 6 in miniature.

7.5. **DPLL in Practice.** The mathematics of Section 7.4 leaves a program three things to settle: how to store the constraints, how to store the valuation as it grows, and how to see quickly which clauses are units and which are falsified. The answers are sets, and an operation on them that throws one switch at a time and simplifies the constraints to match.

Clause sets. The matrix of Section 7.3 already ignored two things a formula records: the order of the columns, and the order and repetition of the entries in a column. Neither changes what the formula says, since $\wedge$ and $\vee$ are commutative, associative and idempotent. A program stores only what matters, and that is a set of sets.

Definition 7.40 (clause set)**.** A clause set records which clauses occur, each as the set of its literals.

Formally, the *clause* of $\ell_1 \vee \cdots \vee \ell_n$ is the set $\{\ell_1, \ldots, \ell_n\}$, a *clause set* is a finite set of clauses, and the *clause set* of $C_1 \wedge \cdots \wedge C_m$ is the set of the clauses of $C_1, \ldots, C_m$. We write C, D for clauses and $\mathcal{S}$ for clause sets.

We still write a clause as the disjunction of its members, so that $p \vee q$, $q \vee p$ and $p \vee q \vee p$ are now the same clause, $\{p, q\}$, and the empty clause $\square$ is the empty set of literals. A valuation satisfies a clause set when it satisfies each of its clauses. Satisfaction, paths and the statuses of a clause depend only on which literals occur, so everything in Section 7.4 holds for clause sets as it stands, a DPLL tree for $\mathcal{S}$ being one for the formula whose clause set is $\mathcal{S}$. From here on the constraints are a clause set $\mathcal{S}$.

Definition 7.41 (clause form)**.** The clause form of a formula is the clause set of its conjunctive normal form.

Formally, $\mathrm{cl}(\varphi)$ is the clause set of $\mathrm{cnf}(\mathrm{nnf}(\varphi))$, and $\mathrm{cl}(\Gamma)$, for a finite set Γ of formulas, is the union of the clause forms of its members.

A valuation satisfies $\mathrm{cl}(\Gamma)$ exactly when it satisfies Γ. For instance, the clause form of the formula of Example 7.22 is

$$\{p \vee r,\ \neg q \vee r\}.$$

Partial valuations as sets. A program stores a partial valuation as the set of literals it makes true,

$$u = \{\ell : u \models \ell\},$$

which is the set we have been writing all along; conversely, a finite set of literals containing no letter together with its negation is a partial valuation. The definitions of Section 7.2 become operations on sets:

$$u \models \ell \quad \text{iff} \quad \ell \in u, \qquad u + \ell = u \cup \{\ell\}, \qquad u \preceq w \quad \text{iff} \quad u \subseteq w.$$

Throwing a switch. Each step of the search throws one switch. It is simpler to throw the switch in the clause set itself than to keep the clause set fixed and look up every literal in u at every step. Turning ℓ on settles every clause that contains ℓ, which can be forgotten, and crosses $\bar{\ell}$ out of every other clause, since $\bar{\ell}$ can no longer help.

Definition 7.42 (restriction)**.** Restricting a clause set to a literal turns the literal on and simplifies the clauses to match.

Formally, for a clause set $\mathcal{S}$ and a literal ℓ, the *restriction* of $\mathcal{S}$ to ℓ is

$$\mathcal{S}|_\ell \;\coloneqq\; \{C \setminus \{\bar{\ell}\} : C \in \mathcal{S} \text{ and } \ell \notin C\}.$$

For literals $\ell_1, \dots, \ell_k$ we write

$$\mathcal{S}|_{\ell_1,\dots,\ell_k} \;\coloneqq\; (\cdots(\mathcal{S}|_{\ell_1})\cdots)|_{\ell_k},$$

and $\mathcal{S}|_u$ for $\mathcal{S}|_{\ell_1,\dots,\ell_k}$ when $u = \{\ell_1, \dots, \ell_k\}$ is a partial valuation.

In words: $\mathcal{S}|_\ell$ is what the constraints still demand once ℓ is true. In the matrix, the columns with a tick disappear and the crossed-out entries are erased. For the five rules of Example 7.15,

$$\mathcal{S}|_p = \{q,\ \neg q \vee r,\ \neg q \vee \neg r\} \qquad \text{and} \qquad \mathcal{S}|_{p,q} = \{r,\ \neg r\}.$$

The notation $\mathcal{S}|_u$ needs the order of the ℓ_i not to matter, and the first part of the next lemma says that it does not.

Lemma 7.43 (restriction)**.** *Let $\mathcal{S}$ be a clause set and u a partial valuation, and write $\bar{u}$ for the set of complements of the members of u.*

(i) In whatever order the members of u are taken,

$$\mathcal{S}|_u = \{C \setminus \bar{u} : C \in \mathcal{S} \text{ and } C \cap u = \varnothing\}.$$

(ii) For every valuation v with $u \preceq v$,

$$v \models \mathcal{S} \qquad \text{iff} \qquad v \models \mathcal{S}|_u.$$

(iii) The partial valuation u is alive for $\mathcal{S}$ if and only if $\mathcal{S}|_u$ is satisfiable.

Proof. Part (i) is by induction on the number of members of u, one at a time: restricting to ℓ deletes the clauses containing ℓ and removes $\bar{\ell}$ from the rest, and the two operations for different letters do not interfere. For (ii), a clause deleted in $\mathcal{S}|_u$ contains a literal of u, which v makes true; and a literal removed from a clause is the complement of a literal of u, which v makes false, so removing it does not change whether v satisfies the clause. For (iii), if $v \models \mathcal{S}$ with $u \preceq v$, then $v \models \mathcal{S}|_u$ by (ii). Conversely, $\mathcal{S}|_u$ mentions no letter that u decides, by (i), so a valuation that satisfies it can be changed on those letters to agree with u and still satisfy it; the changed valuation extends u, and it satisfies $\mathcal{S}$ by (ii). □

In words: throwing the switches of u one at a time, in any order, leaves exactly what remains to be done, and the search may forget what it has already settled.

The three readings of a clause now become three things to look for in $\mathcal{S}|_u$, which is all the procedure below ever inspects. By the first part of the lemma, for a clause C of $\mathcal{S}$:

- C is satisfied under u exactly when it has disappeared from $\mathcal{S}|_u$;
- C is falsified under u exactly when it has become $\square$;
- C is a unit under u with undecided literal ℓ exactly when it has become the unit clause ℓ.

So $u \models \mathcal{S}$ when $\mathcal{S}|_u$ is empty, u is a dead end when $\square \in \mathcal{S}|_u$, and a unit clause of $\mathcal{S}|_u$ names a forced move.

The procedure. In pseudocode, with $\mathcal{S}$ a clause set and u a set of literals:

```
procedure DPLL(S)
    return SEARCH(S, {})                    -- nothing switched yet

procedure SEARCH(S, u)                      -- S is the clause set restricted to u
    F := []                                 -- the forced moves of this call
    while S contains a unit clause l and not the empty clause:
        S := S|l;  u := u + l;  F := F, l   -- a forced move
    if S contains the empty clause: return CLOSED(F; CROSS)
    if S is empty: return OPEN(u)
    p := a letter of S                      -- any letter of S will do
    r := SEARCH(S|p, u + p)
    if r is OPEN(u'): return r
    s := SEARCH(S|not p, u + not p)
    if s is OPEN(u'): return s
    T1, T2 := the trees in r and s          -- both answers are CLOSED
    return CLOSED(F; FORK(p: T1, not p: T2))
```

Here `l` is a literal ℓ, `not p` is $\neg p$, `{}` is $\varnothing$, `[]` is the empty list, `S|l` is $\mathcal{S}|_\ell$, and `u + l` is $u + \ell$, that is, $u \cup \{\ell\}$.

A "no" has to come with its evidence, and the procedure builds it as it goes. The `T` in `CLOSED(T)` is a piece of tree, to be hung below the node at which the call started:

- `F; CROSS` is a chain of nodes carrying the forced literals of `F`, in order, ending in a cross;
- `F; FORK(p: T1, not p: T2)` is the same chain ending in a fork, with a node p above `T1` and a node $\neg p$ above `T2`.

By the reading of $\mathcal{S}|_u$ above, each forced move is an application of *unit*, the fork is an application of *decide*, and `CROSS` and `OPEN` are written exactly when *close* and *finish* apply. So a run grows a DPLL tree for the original clause set, and a `CLOSED` answer hands back the whole of it. On the five rules of Example 7.34, for instance, the call on $\mathcal{S}|_p$ returns `CLOSED(q, r; CROSS)`, the call on $\mathcal{S}|_{\neg p}$ returns `CLOSED(q; CROSS)`, and the top call, which forced nothing, returns

```
CLOSED(FORK(p: (q, r; CROSS), not p: (q; CROSS)))
```

which is the tree drawn there.

Theorem 7.44 (correctness of DPLL)**.** *For every clause set $\mathcal{S}$, the procedure* `DPLL`$(\mathcal{S})$ *stops, and*

- *it returns some* `CLOSED(T)` *if and only if $\mathcal{S}$ is unsatisfiable, and then* `T`*, hung below a root, is a closed DPLL tree for $\mathcal{S}$;*
- *when it returns* `OPEN(u)`*, every valuation v with $u \preceq v$ satisfies $\mathcal{S}$.*

Proof. Each forced move and each recursive call removes a letter from the clause set and adds it to u, so with n letters the loop runs at most n times in each call and the recursion is at most n deep.

An answer `OPEN(u)` comes with $\mathcal{S}|_u$ empty, so $u \models \mathcal{S}$, and every valuation extending u satisfies $\mathcal{S}$ by the lemma on the status of a clause (Lemma 7.30).

For `CLOSED`, show by induction on the depth of the recursion, from the deepest calls up, that a call `SEARCH`$(\mathcal{S}|_u, u)$ that returns `CLOSED(T)` has grown `T` by the rules of DPLL below a branch with partial valuation u, and that every branch of `T` is closed. The literals of `F` are forced moves, by the reading of $\mathcal{S}|_u$ above. If the call ends in `CROSS`, the empty clause it found is a clause of $\mathcal{S}$ falsified on the branch. If it ends in a fork, the two recursive calls returned `CLOSED(T1)` and `CLOSED(T2)`, and by the induction hypothesis `T1` and `T2` are closed pieces grown below p and below $\neg p$. At the top call, $u = \varnothing$, so `T` hung below a root is a closed DPLL tree for $\mathcal{S}$, and $\mathcal{S}$ is unsatisfiable by soundness (Theorem 7.37).

Conversely, if $\mathcal{S}$ is unsatisfiable, no answer `OPEN(u)` is possible, since it would give a valuation that satisfies $\mathcal{S}$; as the procedure stops, it returns `CLOSED`. □

In words: `DPLL` decides satisfiability, and whichever the answer, it hands back the evidence: a setting for yes, a closed tree for no.

Example 7.45 (the platform, cont'd: a setting)**.** The colleague who wrote the rules looks at Harriet's closed tree and admits that the second is wrong: `offline-mode` does not need `fast-sync`. Harriet runs the procedure on $\mathcal{S} = \{C_1, C_3, C_4, C_5\}$. Nothing is forced, so it decides p, and the two sides run

$$\mathcal{S}|_p = \{q,\ \neg q \vee r,\ \neg q \vee \neg r\}, \qquad \mathcal{S}|_{p,q} = \{r,\ \neg r\}, \qquad \mathcal{S}|_{p,q,r} = \{\square\},$$
$$\mathcal{S}|_{\neg p} = \{q\}, \qquad \mathcal{S}|_{\neg p,q} = \varnothing.$$

The first line closes, and the call on $\mathcal{S}|_p$ returns `CLOSED(q, r; CROSS)`; the second empties the clause set, and the procedure returns `OPEN` with $u = \{\neg p, q\}$: `fast-sync` off, `offline-mode` on, and `compression` as you please.

Remark 7.46 (at work: restriction without copying). Written out, $\mathcal{S}|_\ell$ is a new clause set, and a search that may have to go back must keep the old ones too. Solvers do not copy. They keep $\mathcal{S}$ as it is and the literals of u on a stack, and compute what they need of $\mathcal{S}|_u$ on demand. A clause can only become a unit or empty when its last but one live literal is crossed out, so a solver watches two live literals in each clause and looks at the clause only

when one of them is crossed out.[5] The restriction is the mathematics; the watched literals are the bookkeeping that makes it cheap.

Choosing the letter. The procedure leaves one choice open, which letter to decide, and much of the engineering of a solver lies in this *branching heuristic*. On

$$\neg p \vee q, \quad \neg p \vee r, \quad \neg p \vee s, \quad p \vee t, \quad p \vee \neg t$$

deciding p on forces q, r and s and finishes at once, while deciding q, r or s first forces nothing. A common rule of thumb is to decide a letter that occurs in many short clauses. The results above hold for every order; what the order changes is the size of the tree, sometimes enormously.[6]

Easy to check, hard to find. The two answers come with evidence of very different sizes. `OPEN(u)` comes with u, checked by finding a literal of u in each clause. `CLOSED(T)` comes with the tree `T`, checked node by node against Figure 7.2, at a cost proportional to its size, and the tree can be exponential. For some clause sets it must be: the optional Section 7.10 shows that for the pigeonhole sets, which say that $n + 1$ pigeons do not fit into n holes, every closed tree is exponential in n.

The asymmetry is the shape of the problem, not a fault of the method. Deciding whether a clause set is satisfiable is the problem SAT, the first problem shown to be *NP-complete*.[7] That means two things. Its "yes" answers can be checked quickly. And every problem whose "yes" answers can be checked quickly reduces to it quickly. ("Quickly" means in a number of steps bounded by a polynomial in the size of the input.) The problems whose "yes" answers can be checked quickly form the class NP, and those that can be decided quickly the class P. Whether SAT can be decided quickly is the question whether P = NP, and nobody knows. What is known is that the procedure of this chapter decides it, that solvers built on it settle instances with millions of switches, and that no method is known to be fast on every instance.

Remark 7.47 (three strategies). Three habits of the procedure are worth keeping whenever a problem is a search among finitely many choices.

- *Propagate before you branch.* A forced move costs a lookup and loses nothing; a decision may double the work below it.
- *Encode by forbidding.* Ask what must never happen and write one clause for each forbidden pattern; "exactly one of these" is easy as "not none, and never two".

[5] The two watched literals are due to Moskewicz et al. (2001). The first of the two DPLL papers stated its rules as operations on the clause set, in the manner of restriction.

[6] Exercise 7.8 gives clause sets on which one order closes the tree in four nodes and another needs more than 2^k branches, and Example 7.99, in the optional Section 7.10, shows the same effect at scale. Modern solvers choose each letter as the search goes, from what it has seen. The best known heuristic, VSIDS, raises the score of the letters in each newly learned clause and every so often scales all scores down, so that recent failures count most (Moskewicz et al., 2001). The first of the two DPLL papers had a further rule, the *pure-literal* rule, which solvers still use when preparing their input; Exercise 7.7 shows what it keeps and what it loses.

[7] By Cook (1971), and independently by Levin (1973).

- *Ask what a failure tells you.* A closed branch proves that the literals on it cannot hold together, a fact the constraints imply but do not state (Remark 7.35); write it down and never pay for it twice.

Remark 7.48 (SAT and SMT). Questions about programs and hardware are often about numbers, arrays and bit-strings, where an atomic formula such as $x + 1 > y$ is true or false according to the values of x and y. An *SMT solver*, for satisfiability modulo theories, reads each atomic formula as a letter and runs `DPLL`; whenever the search has set some atomic formulas, it asks a decision procedure for the theory (arithmetic, arrays, equality) whether they can hold together, and if not, as with $x > 1$ and $x < 0$, adds the reason as a new clause. The engine is the procedure of this section.

7.6. **From Sentences to Clauses.** The propositional problem is solved: a question without quantifiers becomes a clause set, and `DPLL` decides it, with a certificate either way. The questions we started from are not of that kind. Their constraints say "every" and "some", and no setting of finitely many switches says either. The next three sections bring a first-order question down to propositional ones, in three moves.

1. *Put the quantifiers in one place.* Each sentence is rewritten, without changing what it says, so that all its quantifiers stand in a row at the front.
2. *Name the existentials.* Each "some" is replaced by a name for its witness, a new function of the universal variables before it, as a search by hand names the someone it is looking for. Only universal quantifiers are left, and the sentence becomes a set of clauses with variables in them.
3. *Instantiate the universals.* The clauses are tested at the names, one instance at a time, and each instance is a clause over switches (Section 7.7). The instances go to `DPLL` level by level (Section 7.8).

The first two moves are routine rewriting, and this section treats them as a black box: what goes in, what comes out, and what is guaranteed (the optional Section 7.9 opens the box). The third move is where the mathematics is, and it takes the next two sections. From here on the formulas are again the first-order formulas of Chapter 6, over a signature Σ.

Remark 7.49 (a second route: SMT). Removing the quantifiers first is a choice. An SMT solver (Remark 7.48) keeps them and deals with them on the side, adding instances of the universal sentences to its propositional problem as the search goes. Removing them first has an advantage: a single theorem, Herbrand's, then explains why instances are enough. The two routes meet in Section 7.8.

What comes out of the box is a set of clauses with variables in them, and the first thing to settle is what such a clause means.

Definition 7.50 (first-order clause)**.** A first-order clause is built like a propositional clause, but its literals are atomic formulas about terms, and the terms may contain variables.

Formally, a *clause* over Σ is a finite set of literals, that is, of atomic Σ-formulas and their negations, possibly with variables, and a *clause set* is a finite set of clauses. A clause with no variables is a *ground* clause.

For instance, of the two clauses

$$\neg P(x) \vee Q(f(x)) \qquad \text{and} \qquad \neg P(c) \vee Q(f(c))$$

the second is ground.

Definition 7.51 (satisfying a first-order clause set)**.** A structure satisfies a clause when the clause holds of everybody: whatever its variables stand for, one of its literals is true.

Formally, let $\mathfrak{M}$ be a Σ-structure and C a clause over Σ. Then $\mathfrak{M} \models C$ if

$$\text{for every assignment } s, \qquad \mathfrak{M}, s \models \ell \text{ for some } \ell \in C,$$

and $\mathfrak{M} \models \mathcal{S}$, for a clause set $\mathcal{S}$, if $\mathfrak{M} \models C$ for every $C \in \mathcal{S}$.

In words: a clause is read as the universal closure of the disjunction of its literals. For a clause $C = \ell_1 \vee \cdots \vee \ell_n$ with variables $x_1, \ldots, x_k$,

$$\mathfrak{M} \models C \qquad \text{exactly when} \qquad \mathfrak{M} \models \forall x_1 \cdots \forall x_k \, (\ell_1 \vee \cdots \vee \ell_n).$$

In particular, for a ground clause C, $\mathfrak{M} \models C$ exactly when $v_{\mathfrak{M}} \models C$ in the sense of Section 7.3, where $v_{\mathfrak{M}}$ is the valuation that gives each atomic sentence its truth value in $\mathfrak{M}$. One warning, which matters later: each clause binds its own variables, so the x of one clause has nothing to do with the x of another, and a variable may be renamed within a clause at will.

The box adds symbols to the signature. If Σ^+ is Σ with some symbols added, a Σ^+-structure $\mathfrak{M} = (\mathbb{D}, I)$ has a *restriction* to Σ,

$$\mathfrak{M}{\restriction}\Sigma \;=\; (\mathbb{D}, I{\restriction}\Sigma),$$

the same domain with the added symbols forgotten, as in the completeness proof of Chapter 6. A Σ-sentence mentions none of the added symbols, so it is true in $\mathfrak{M}$ exactly when it is true in $\mathfrak{M}{\restriction}\Sigma$.

Theorem 7.52 (the clause form)**.** *There is an algorithm that takes each finite set Γ of Σ-sentences to a clause set $\mathrm{cl}(\Gamma)$ over a signature Σ^+, which is Σ with new function and constant symbols added, such that*

(i) Γ is satisfiable if and only if $\mathrm{cl}(\Gamma)$ is;
(ii) if $\mathfrak{M} \models \mathrm{cl}(\Gamma)$, then $\mathfrak{M}{\restriction}\Sigma \models \Gamma$.

The clause set $\mathrm{cl}(\Gamma)$ that the algorithm returns is called the *clause form* of Γ. In words: the clause form keeps the verdict, by (i), and comes with a way back, by (ii): forget the new symbols. It does not keep the meaning, since it speaks of symbols that Γ does not have, so Γ and $\mathrm{cl}(\Gamma)$ are equisatisfiable rather than true in the same structures.

The new symbols name the witnesses of the existential quantifiers, and are called *Skolem functions* and *Skolem constants*. For instance, "everyone has someone who earns more", $\forall x \, \exists y \, \mathsf{Richer}(y, x)$, becomes the single clause

$$\mathsf{Richer}(f(x), x),$$

with $f(x)$ naming whoever earns more than x, over $\Sigma^+ = \Sigma \cup \{f\}$; and "someone earns more than everyone", $\exists y \, \forall x \, \mathsf{Richer}(y, x)$, becomes $\mathsf{Richer}(a, x)$ over $\Sigma \cup \{a\}$, with the constant a naming that someone. The optional Section 7.9 gives the algorithm and the proof; for the problem sheets and the rest of the chapter, the theorem is all that is needed.

Example 7.53 (the company, cont'd: the policy as clauses)**.** To decide whether $\{\mathrm{P}_1, \mathrm{P}_2\} \models \mathrm{C}$, Harriet needs the clause form of $\{\mathrm{P}_1, \mathrm{P}_2, \neg\mathrm{C}\}$, a set of sentences over the company signature Σ. The two rules have no existential quantifier, and $\neg\mathrm{C}$ says that some engineer reports to an engineer; its witness gets the Skolem constant a, the name her first search in Example 7.4 gave it. So $\Sigma^+ = \Sigma \cup \{a\}$, and the clause form is four clauses:

$$C_1 = \neg\mathsf{Eng}(x) \vee \mathsf{Mgr}(\mathsf{boss}(x)) \qquad \text{an engineer's boss is a manager;}$$
$$C_2 = \neg\mathsf{Mgr}(x) \vee \neg\mathsf{Eng}(x) \qquad \text{no manager is an engineer;}$$
$$C_3 = \mathsf{Eng}(a) \qquad a \text{ is an engineer;}$$
$$C_4 = \mathsf{Eng}(\mathsf{boss}(a)) \qquad \text{so is } a\text{'s boss.}$$

By the reduction lemma (Lemma 7.3) and the theorem on the clause form (Theorem 7.52),

$$\{\mathrm{P}_1, \mathrm{P}_2\} \models \mathrm{C} \qquad \text{if and only if} \qquad \{C_1, C_2, C_3, C_4\} \text{ is unsatisfiable.}$$

The colleague's weaker policy is $\{C_1, C_3, C_4\}$, which is satisfiable: the two-person company $\mathfrak{M}$ of Example 7.4, with a naming the engineer who reports to an engineer, has $\mathfrak{M} \models \{C_1, C_3, C_4\}$, and $\mathfrak{M}{\restriction}\Sigma$ is the counter-model of that example.

Remark 7.54 (equality). The symbol $=$ is a relation symbol like any other, and in a clause set it is treated as one: nothing in the definitions above knows that it means identity. A set of sentences that is to reason about identity must say what identity is, through the equality axioms, and so must a clause set. Those axioms are infinitely many, one for every formula; for a clause set over a finite signature, finitely many clauses do the same work, and we call them the *equality clauses*. They are reflexivity, symmetry, transitivity, and one *congruence* clause for each argument place of each function symbol and of each relation symbol other than $=$, saying that equals may be put for equals in that place. For $=$ itself no congruence clause is needed: symmetry and transitivity already do the work. For a signature with a one-place function symbol f, one-place relation symbols P and Q, and a two-place relation symbol R besides $=$, the congruence clauses are

$$\neg(x = y) \vee f(x) = f(y),$$
$$\neg(x = y) \vee \neg P(x) \vee P(y), \qquad \neg(x = y) \vee \neg Q(x) \vee Q(y),$$
$$\neg(x = y) \vee \neg R(x, z) \vee R(y, z),$$
$$\neg(x = y) \vee \neg R(z, x) \vee R(z, y).$$

Exercise 7.12 writes out the rest and puts a question about identity through the machinery of this chapter. A structure that satisfies such a clause set interprets $=$ as some relation that behaves like identity but need not be it. A remark in the next section shows how to recover from it a structure in which $=$ *is* identity.

7.7. **Herbrand's Theorem.** After the first two moves, the question $\Gamma \models \varphi$ has become whether a set of clauses with variables, all read universally, is unsatisfiable. A clause with a variable speaks of every element of every structure (five people, the natural numbers, the real line), and no solver can look at every structure. A search by hand, such as the one in Example 7.4,

does something much smaller: it looks only at the things it can name. This section proves that this is enough, in two steps.

- If a clause set is satisfied by any structure at all, it is satisfied by one made of names, and such a structure is a setting of the switches, one for each atomic sentence about the names.
- If it has no model, finitely many of its instances at those names already show it.

The names are built from the symbols that the clauses actually use. The signature may have others (the clauses of Example 7.53 mention neither ceo nor Richer), but symbols a clause set does not mention make no difference to whether it is satisfiable: a structure can be restricted to the smaller signature, and expanded back by giving the other symbols any meaning.

Definition 7.55 (the signature of a clause set)**.** The signature of a clause set is the vocabulary it actually uses.

Formally, the *signature* $\Sigma_{\mathcal{S}}$ of a clause set $\mathcal{S}$ consists of the constant, function and relation symbols that occur in $\mathcal{S}$, with one new constant symbol added if no constant occurs in $\mathcal{S}$.

The extra constant is needed because a signature without constants names nothing, and the names are to serve as the domain of a structure, which must not be empty. A structure $\mathfrak{M}$ for a larger signature satisfies $\mathcal{S}$ exactly when its restriction $\mathfrak{M}{\restriction}\Sigma_{\mathcal{S}}$ does (with the added constant, if there is one, naming any element).

Definition 7.56 (Herbrand universe)**.** The Herbrand universe of a clause set is the set of everything the clause set can name.

Formally, the *Herbrand universe* $\mathrm{H}(\mathcal{S})$ of a clause set $\mathcal{S}$ is the set of closed $\Sigma_{\mathcal{S}}$-terms.

In words: start from the constants and apply the function symbols in every possible way. For a clause set whose only constant is c and whose only function symbol is a one-place f, the universe is c, $f(c)$, $f(f(c))$, and so on.

Definition 7.57 (level)**.** The level of a closed term is the depth to which function symbols are nested in it.

Formally, the *level* of a closed term t is defined by recursion on the shape of t:

$$\mathrm{level}(t) \coloneqq \begin{cases} 0 & \text{if } t \text{ is a constant,} \\ 1 + \max_i \mathrm{level}(t_i) & \text{if } t \text{ is } f(t_1, \ldots, t_n). \end{cases}$$

For a clause set $\mathcal{S}$ and $k \geq 0$, we write $\mathrm{H}_k(\mathcal{S})$ for the set of terms of $\mathrm{H}(\mathcal{S})$ of level at most k.

For instance, $f(f(c))$ has level 2, and in the universe above $\mathrm{H}_1 = \{c, f(c)\}$.

Measuring terms by depth is a choice; any measure would do for which there are finitely many terms of each size, and depth is the simplest. What matters is that every $\mathrm{H}_k(\mathcal{S})$ is finite and each contains the one before,

$$\mathrm{H}_0(\mathcal{S}) \subseteq \mathrm{H}_1(\mathcal{S}) \subseteq \mathrm{H}_2(\mathcal{S}) \subseteq \cdots,$$

while together they make up the whole universe, which is infinite as soon as there is a function symbol. A search must therefore work through the

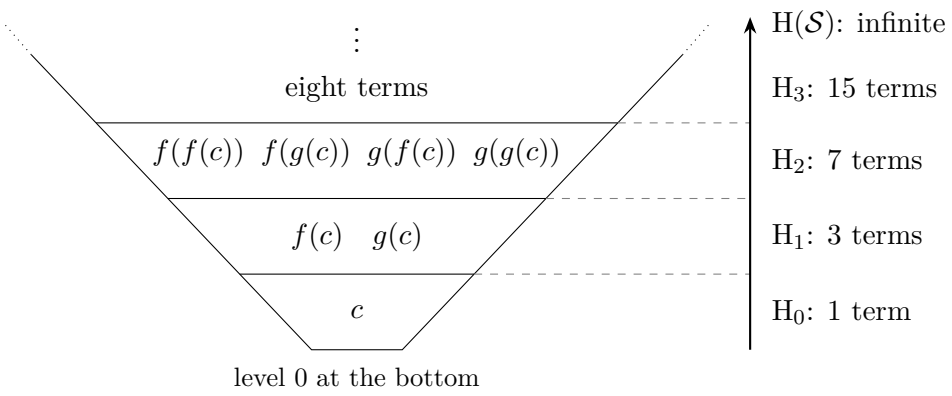


FIGURE 7.3. The Herbrand universe, level by level.

universe one level at a time. Figure 7.3 shows how fast the levels grow with a constant c and two one-place function symbols: each level holds twice as many terms as the one below.

Definition 7.58 (Herbrand base)**.** The Herbrand base of a clause set is the set of atomic sentences it can form about its names: the switches of a first-order search.

Formally, the *Herbrand base* of a clause set $\mathcal{S}$ is the set of atomic $\Sigma_{\mathcal{S}}$-sentences $R(t_1, \ldots, t_n)$, with R an n-place relation symbol of $\Sigma_{\mathcal{S}}$ and $t_1, \ldots, t_n \in \mathrm{H}(\mathcal{S})$.

For instance, if $\Sigma_{\mathcal{S}}$ above has one relation symbol, a one-place P, the base is $P(c)$, $P(f(c))$, $P(f(f(c)))$, and so on.

Definition 7.59 (Herbrand structure, Herbrand model)**.** A Herbrand structure is a structure whose elements are names, each naming itself.

Formally, let $\mathcal{S}$ be a clause set. A *Herbrand structure* for $\mathcal{S}$ is a $\Sigma_{\mathcal{S}}$-structure $\mathfrak{H} = (\mathrm{H}(\mathcal{S}), I)$ in which

- $I(c) = c$ for each constant symbol c;
- $I(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for each n-place function symbol f;
- $I(R)$ is any set of n-tuples of closed terms, for each n-place relation symbol R.

A *Herbrand model* of $\mathcal{S}$ is a Herbrand structure $\mathfrak{H}$ for $\mathcal{S}$ that is a model of $\mathcal{S}$, that is, with $\mathfrak{H} \models \mathcal{S}$.

In words: the domain is the names, the constants and function symbols do nothing but build names, and all the freedom is in the relations. So a Herbrand structure is nothing more than a choice of true or false for each atomic sentence of the base: *a Herbrand structure is a valuation of the Herbrand base*, a setting of one switch for each atomic sentence about the names. A search for a Herbrand model is a search for a setting.

Remark 7.60 (looking back: the term structure). The completeness proof of Chapter 6 built its structure out of closed terms, each naming itself, with

the relations chosen by a maximal consistent set. That structure was a Herbrand structure, for the signature with the witnesses added, and the truth lemma said that it satisfied the set. Nothing in this section is new in kind; what is new is that the relations are now chosen by a search rather than by a construction that lists every sentence.

Example 7.61 (the company, cont'd: the names of the policy)**.** The clause set $\mathcal{S} = \{C_1, C_2, C_3, C_4\}$ of Example 7.53 is over Σ^+, the company signature with a added, but it uses only four of its symbols: $\Sigma_{\mathcal{S}}$ has the constant a, the function symbol boss and the relation symbols Eng and Mgr, and no ceo or Richer. Its Herbrand universe is

$$a,\ \mathsf{boss}(a),\ \mathsf{boss}(\mathsf{boss}(a)),\ \mathsf{boss}(\mathsf{boss}(\mathsf{boss}(a))),\ \ldots,$$

one term for each level: $\mathrm{H}_0 = \{a\}$, $\mathrm{H}_1 = \{a, \mathsf{boss}(a)\}$, and so on. It is infinite, because boss can be applied again to its own result, and in a Herbrand structure the chain of bosses never comes back on itself. The base is the atomic sentences $\mathsf{Eng}(t)$ and $\mathsf{Mgr}(t)$ for t in the universe. If a Herbrand structure $\mathfrak{H}$ makes every $\mathsf{Eng}(t)$ and every $\mathsf{Mgr}(t)$ true, for instance, then $\mathfrak{H} \models \{C_1, C_3, C_4\}$ and $\mathfrak{H} \not\models C_2$.

Instances. To say what a clause demands of a Herbrand structure, we put names in for its variables, several variables at once. The tool for that is a substitution.

Definition 7.62 (substitution)**.** A substitution is a finite list of instructions, each saying which term to put in for one variable.

Formally, a *substitution* is a finite list $\sigma = [t_1/x_1, \ldots, t_n/x_n]$ of pairs, each of a term t_i and a variable x_i, in which the variables $x_1, \ldots, x_n$ are distinct. We also write it $[\bar{t}/\bar{x}]$.

For instance, $[\mathsf{boss}(y)/x, \mathsf{ceo}/z]$ says: put $\mathsf{boss}(y)$ for x and ceo for z. It says nothing about y, which it leaves alone. The empty list $[\,]$ is a substitution too, and changes nothing.

Substitutions are applied to terms, atomic formulas, literals and clauses, which for short we call *expressions*.

Definition 7.63 (applying a substitution)**.** Applying a substitution to an expression puts the terms in for the variables, everywhere and all at once.

Formally, let $\sigma = [t_1/x_1, \ldots, t_n/x_n]$ be a substitution. For an expression e, the expression $e\sigma$ is defined by recursion on the shape of e:

$$e\sigma \coloneqq \begin{cases} t_i & \text{if } e \text{ is the variable } x_i, \\ y & \text{if } e \text{ is a variable } y \text{ other than } x_1, \ldots, x_n, \\ f(u_1\sigma, \ldots, u_m\sigma) & \text{if } e \text{ is } f(u_1, \ldots, u_m), \\ R(u_1\sigma, \ldots, u_m\sigma) & \text{if } e \text{ is } R(u_1, \ldots, u_m), \\ \neg(A\sigma) & \text{if } e \text{ is } \neg A, \text{ with } A \text{ atomic}, \\ \{\ell_1\sigma, \ldots, \ell_m\sigma\} & \text{if } e \text{ is the clause } \{\ell_1, \ldots, \ell_m\}. \end{cases}$$

In the third line f may be a constant, with $m = 0$, and then $e\sigma$ is e. In words: every occurrence of each x_i is replaced by t_i, and the terms put in are not looked at again. That is what "all at once" means: $\mathsf{Richer}(x, y)[y/x, x/y]$

is $\mathsf{Richer}(y, x)$, whereas putting y for x and then x for y, one after the other, would give $\mathsf{Richer}(x, x)$. An expression has no quantifiers, so nothing in it can capture a variable, and no renaming is ever needed.

Remark 7.64 (looking back: substitution for one variable). For one variable, $e[t/x]$ is the substitution $\varphi[t/x]$ of Chapter 6, restricted to formulas without quantifiers. Its side conditions, which stop a quantifier of φ from capturing a variable of t, never arise here.

Definition 7.65 (instance, ground instance)**.** An instance of a clause is what the clause says about particular terms, and a ground instance is what it says about particular names.

Formally, let C be a clause. An *instance* of C is a clause $C\sigma$, for a substitution σ. If C is a clause of a clause set $\mathcal{S}$, with variables $x_1, \ldots, x_n$, a *ground instance* of C is a clause $C[t_1/x_1, \ldots, t_n/x_n]$ with $t_1, \ldots, t_n \in \mathrm{H}(\mathcal{S})$.

For instance, the clause $\neg P(x) \vee Q(f(x), y)$ has the instance $\neg P(f(z)) \vee Q(f(f(z)), y)$, by $[f(z)/x]$; in a clause set with the constant c it has the ground instance $\neg P(c) \vee Q(f(c), c)$, by $[c/x, c/y]$. A ground instance is a ground clause, since every variable of C has been given a closed term.

Definition 7.66 (the ground instances of a clause set)**.** The ground instances of a clause set are everything its clauses say about its names, and those at level k use only names of level at most k.

Formally, let $\mathcal{S}$ be a clause set. We write $\mathrm{Gr}(\mathcal{S})$ for the set of all ground instances of all clauses of $\mathcal{S}$, and, for $k \geq 0$, $\mathrm{Gr}_k(\mathcal{S})$ for the set of those ground instances $C[t_1/x_1, \ldots, t_n/x_n]$, with $C \in \mathcal{S}$ and $x_1, \ldots, x_n$ its variables, in which $t_1, \ldots, t_n \in \mathrm{H}_k(\mathcal{S})$.

Each $\mathrm{Gr}_k(\mathcal{S})$ is a finite clause set over the Herbrand base, and $\mathrm{Gr}(\mathcal{S})$ is their union, infinite as soon as the universe is and some clause has a variable. The level bounds the terms put in for the variables, not the terms that end up in the clause: a clause may carry closed terms of its own, and a term put in may land inside a function symbol. So $\mathrm{Gr}_0(\mathcal{S})$ can mention terms of any level; in Example 7.67, Gr_0 contains $\mathsf{Eng}(\mathsf{boss}(a))$ although $\mathrm{H}_0 = \{a\}$.

Example 7.67 (the company, cont'd: the ground instances of the policy)**.** For the four clauses of Example 7.53, Gr_0 has four clauses, the two rules instantiated at a and the two units; Gr_1 has six:

$$\neg\mathsf{Eng}(a) \vee \mathsf{Mgr}(\mathsf{boss}(a)), \qquad \neg\mathsf{Eng}(\mathsf{boss}(a)) \vee \mathsf{Mgr}(\mathsf{boss}(\mathsf{boss}(a))),$$
$$\neg\mathsf{Mgr}(a) \vee \neg\mathsf{Eng}(a), \qquad \neg\mathsf{Mgr}(\mathsf{boss}(a)) \vee \neg\mathsf{Eng}(\mathsf{boss}(a)),$$
$$\mathsf{Eng}(a), \qquad \mathsf{Eng}(\mathsf{boss}(a)).$$

Lemma 7.68 (a Herbrand structure is a valuation)**.** *For a Herbrand structure $\mathfrak{H}$ for $\mathcal{S}$, write $v_{\mathfrak{H}}$ for the valuation of the Herbrand base that makes an atomic sentence true exactly when it is true in $\mathfrak{H}$. Then*

$$\mathfrak{H} \models \mathcal{S} \qquad \textit{if and only if} \qquad v_{\mathfrak{H}} \models \mathrm{Gr}(\mathcal{S}).$$

Proof. First, every closed term t names itself, $\mathrm{Val}(t) = t$: by induction on t, a constant names itself by the first item of the definition of a Herbrand structure, and $f(t_1, \ldots, t_n)$ names $I(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ by the second and the induction hypothesis.

Now let C be a clause of $\mathcal{S}$ with variables $\bar{x}$. An assignment s into $\mathfrak{H}$ gives each x_i a closed term $s(x_i)$, which names itself; so, by the substitution lemma, a literal of C is true under s exactly when the literal of $C[s(\bar{x})/\bar{x}]$ is true in $\mathfrak{H}$ outright. Every ground instance of C is $C[s(\bar{x})/\bar{x}]$ for some assignment s, the one that sends each x_i to the term put in for it, so $\mathfrak{H} \models C$ exactly when every ground instance of C has a literal true in $\mathfrak{H}$. A ground literal is true in $\mathfrak{H}$ exactly when $v_{\mathfrak{H}}$ makes it true, by the atomic clause of the definition of satisfaction and the definition of $v_{\mathfrak{H}}$; so $\mathfrak{H} \models C$ exactly when $v_{\mathfrak{H}} \models C'$ for every ground instance C' of C. □

In words: to say what a Herbrand structure does to a clause set is to run through the ground instances, and that is a propositional matter.

Lemma 7.69 (Herbrand models suffice)**.** *A clause set is satisfiable if and only if it has a Herbrand model.*

Proof. A Herbrand model is a model, so only the forward direction needs an argument. Let $\mathfrak{M} \models \mathcal{S}$. By the paragraph after the definition of $\Sigma_{\mathcal{S}}$ we may take $\mathfrak{M}$ to be a $\Sigma_{\mathcal{S}}$-structure. Let $\mathfrak{H}$ be the Herbrand structure in which an atomic sentence $R(t_1, \ldots, t_n)$ of the base is true exactly when it is true in $\mathfrak{M}$. By the lemma it is enough to show that $v_{\mathfrak{H}} \models C[\bar{t}/\bar{x}]$ for every ground instance of every clause C of $\mathcal{S}$.

Since $\mathfrak{M} \models C$, some literal of C is true in $\mathfrak{M}$ under the assignment s that sends each x_i to the element $\mathrm{Val}(t_i)$ of $\mathfrak{M}$; by the substitution lemma, that literal with $\bar{t}$ put in is true in $\mathfrak{M}$ outright. It is a literal of $C[\bar{t}/\bar{x}]$, a ground literal, and by the definition of $\mathfrak{H}$ it is true in $\mathfrak{H}$, that is, $v_{\mathfrak{H}}$ makes it true. So $v_{\mathfrak{H}} \models C[\bar{t}/\bar{x}]$, hence $\mathfrak{H} \models \mathcal{S}$, and $\mathfrak{H}$ is a Herbrand model of $\mathcal{S}$. □

In words: given a structure that satisfies the clause set, read off which atomic sentences about names it makes true and forget everything else; what is left is a structure made of names that satisfies it too. The reason is that clauses are universal: what holds of every element of $\mathfrak{M}$ holds in particular of the elements that terms name, and those are all a Herbrand structure keeps.

So a search for a counter-model may confine itself to the things it can name: if a counter-model exists at all, one exists that is made of names. One more step turns "among those" into "among finitely many of those", and it needs a consequence of the completeness of NK.

Lemma 7.70 (compactness)**.** *Let* Γ *be a set of sentences over a countable signature. If every finite subset of* Γ *has a model, then* Γ *has a model.*

Proof. We prove the contrapositive: if Γ has no model, some finite subset has none. Suppose Γ has no model. Then $\Gamma \models \bot$, since there is no model of Γ in which $\bot$ fails, and by the completeness of NK there is a derivation of $\bot$ from Γ. A derivation is a finite tree, so it has finitely many open hypotheses, which form a finite subset Δ of Γ with $\Delta \vdash_{\mathrm{NK}} \bot$. By soundness $\Delta \models \bot$, so Δ has no model. □

In words: an infinite set of demands can only fail for a finite reason, and the reason is that proofs are finite.

Theorem 7.71 (Herbrand's theorem)**.** *For every clause set $\mathcal{S}$, the following are equivalent:*

(i) there is no structure $\mathfrak{M}$ with $\mathfrak{M} \models \mathcal{S}$;

(ii) there is a finite $\Delta \subseteq \mathrm{Gr}(\mathcal{S})$ with no valuation $v \models \Delta$, reading Δ as a propositional clause set over the Herbrand base;

(iii) there is a k with no valuation $v \models \mathrm{Gr}_k(\mathcal{S})$.

Proof. One implication at a time.

- *(ii) implies (i).* Suppose $\mathfrak{M} \models \mathcal{S}$. By the lemma that Herbrand models suffice (Lemma 7.69) there is a Herbrand structure $\mathfrak{H} \models \mathcal{S}$, and by the lemma before it $v_{\mathfrak{H}} \models \mathrm{Gr}(\mathcal{S})$, so $v_{\mathfrak{H}} \models \Delta$ for every $\Delta \subseteq \mathrm{Gr}(\mathcal{S})$.
- *(i) implies (ii).* The ground clauses of $\mathrm{Gr}(\mathcal{S})$, each read as the disjunction of its literals, are sentences over a finite, hence countable, signature, and no structure satisfies them all: the truth values of the ground atomic sentences in such a structure would give a valuation $v \models \mathrm{Gr}(\mathcal{S})$, hence a Herbrand model of $\mathcal{S}$ by the lemma that a Herbrand structure is a valuation (Lemma 7.68). By compactness (Lemma 7.70) some finite $\Delta \subseteq \mathrm{Gr}(\mathcal{S})$ has no model, and by the same argument no valuation $v \models \Delta$.
- *(ii) and (iii) are equivalent.* A finite $\Delta \subseteq \mathrm{Gr}(\mathcal{S})$ uses finitely many closed terms, and if k is the largest of their levels, $\Delta \subseteq \mathrm{Gr}_k(\mathcal{S})$; a valuation that satisfies $\mathrm{Gr}_k(\mathcal{S})$ satisfies Δ. And $\mathrm{Gr}_k(\mathcal{S})$ is itself a finite subset of $\mathrm{Gr}(\mathcal{S})$.

□

In words: a set of constraints about everybody has no model exactly when finitely many of its instances, at finitely many names, already clash.

Example 7.72 (the company, cont'd: six clauses)**.** No valuation satisfies Gr_1, the six clauses of Example 7.67, and unit propagation alone shows it:

$\mathsf{Eng}(a)$	the unit clause,
$\mathsf{Mgr}(\mathsf{boss}(a))$	by $\neg\mathsf{Eng}(a) \vee \mathsf{Mgr}(\mathsf{boss}(a))$,
$\mathsf{Eng}(\mathsf{boss}(a))$	the unit clause,
$\times$	$\neg\mathsf{Mgr}(\mathsf{boss}(a)) \vee \neg\mathsf{Eng}(\mathsf{boss}(a))$ is falsified.

These are the lines of Harriet's first search, one for one: a is an engineer, so her boss is a manager, so her boss is not an engineer, but her boss is. By Herbrand's theorem (Theorem 7.71) $\{C_1, \ldots, C_4\}$ is unsatisfiable, so by the reduction lemma (Lemma 7.3) and the theorem on the clause form (Theorem 7.52), $\{\mathrm{P}_1, \mathrm{P}_2\} \models \mathrm{C}$.

The four clauses of Gr_0 do not suffice: the valuation that makes $\mathsf{Eng}(a)$, $\mathsf{Eng}(\mathsf{boss}(a))$ and $\mathsf{Mgr}(\mathsf{boss}(a))$ true and $\mathsf{Mgr}(a)$ false satisfies them. The clash needs the second rule at $\mathsf{boss}(a)$, one level up, which is the line of her search that applied P_2 to a's boss.

What the theorem does not say. Herbrand's theorem (Theorem 7.71) gives no bound on the level at which the clash appears, and the level needed can be large. Nor does it say anything when no clash appears. If $\mathrm{Gr}_0, \mathrm{Gr}_1, \ldots, \mathrm{Gr}_{100}$

are each satisfiable, the theorem does not say that $\mathcal{S}$ is, and it cannot. For the clause set

$$P(c), \qquad \neg P(x) \vee P(f(x))$$

every Gr_k is satisfiable, and so is the set; for another set, every Gr_k up to some large k may be satisfiable and the next one not.

The trouble is the evidence. For a satisfiable set the evidence has to be a structure that satisfies it, and a valuation that satisfies the instances $\mathrm{Gr}_k(\mathcal{S})$ of a single level is not one: it is a choice of truth values for the atomic sentences that occur in $\mathrm{Gr}_k(\mathcal{S})$, and says nothing about the instances above. Can any finite amount of work ever certify that a clause set is satisfiable? This chapter raises the question and does not settle it.

Remark 7.73 (equality in a Herbrand structure). When $=$ is among the symbols and the equality clauses of Remark 7.54 are in $\mathcal{S}$, a Herbrand structure that satisfies $\mathcal{S}$ interprets $=$ as some relation E on the closed terms that is reflexive, symmetric, transitive and respected by every symbol, which is all the clauses say. But E need not be identity: $f(c) = c$ may hold in it while $f(c)$ and c are different terms. To obtain a structure in which $=$ is identity, identify the terms that E relates: the elements are the classes of terms, and the congruence clauses are exactly what makes the functions and relations well defined on the classes. The completeness proof of Chapter 6 makes the same move, for the same reason.

7.8. **A Procedure for First-Order Logic.** This section puts the pieces together into a program, and then asks how good it is. `DPLL` decides one finite clause set, and a first-order clause set has infinitely many ground instances. But by Herbrand's theorem (Theorem 7.71) a clause set $\mathcal{S}$ is unsatisfiable exactly when some $\mathrm{Gr}_k(\mathcal{S})$ is, and each $\mathrm{Gr}_k(\mathcal{S})$ is a finite clause set whose letters are the atomic sentences occurring in it: just what `DPLL` decides. So we try the levels in turn:

```
procedure REFUTE(S)                    -- S a clause set, possibly with variables
    k := 0
    loop:
        if DPLL(GROUND(S, k)) is CLOSED(T): return UNSAT  -- the instances at level k
        k := k + 1
```

The procedure has no other exit, and that is not an oversight.

Theorem 7.74 (the first-order procedure)**.** *For every clause set $\mathcal{S}$, the call* `REFUTE`$(\mathcal{S})$

(i) returns `UNSAT` *if $\mathcal{S}$ is unsatisfiable;*

(ii) runs for ever if $\mathcal{S}$ is satisfiable.

Proof. If $\mathcal{S}$ is unsatisfiable, some $\mathrm{Gr}_k(\mathcal{S})$ is unsatisfiable by Herbrand's theorem (Theorem 7.71), and by the correctness of DPLL (Theorem 7.44) the call `DPLL`$(\mathrm{Gr}_k(\mathcal{S}))$ returns `CLOSED` at the latest at that k, each earlier call having stopped by termination (Lemma 7.36). If $\mathcal{S}$ is satisfiable, some Herbrand structure $\mathfrak{H}$ satisfies it, and $v_{\mathfrak{H}}$ satisfies every $\mathrm{Gr}_k(\mathcal{S})$, so by soundness (Theorem 7.37) no call returns `CLOSED` and `REFUTE` never stops. □

In words: the procedure is a complete method for "no", and no method at all for "yes". Procedures of this kind have a name.

Policy	Level	Ground clauses	Decisions	Verdict
$\{C_1, C_2, C_3, C_4\}$	0	4	0	`OPEN`
	1	6	0	`CLOSED`, so `UNSAT`
$\{C_1, C_3, C_4\}$	0	3	0	`OPEN`
	1	4	0	`OPEN`
	2	5	1	`OPEN`
	...			`OPEN`, for ever

FIGURE 7.4. The policy by machine, level by level.

Definition 7.75 (semi-decision procedure)**.** A semi-decision procedure for a property answers "yes" on every input that has the property, and never on one that lacks it.

Formally, let A be a set of inputs. A procedure is a *semi-decision procedure* for A if, for every input x, it stops and answers that x is in A when $x \in A$, and never gives that answer when $x \notin A$.

So `REFUTE` is a semi-decision procedure for the set of unsatisfiable clause sets. Through the reduction lemma (Lemma 7.3) and the clause form, this gives a way of certifying in finite time every first-order entailment from finitely many premisses that holds, and nothing for one that fails.

Example 7.76 (the company, cont'd: the policy, by machine)**.** Harriet gives the four clauses of Example 7.53 to `REFUTE`. At level 0 the four ground clauses of Example 7.72 go to `DPLL`, which propagates $\mathsf{Eng}(a)$, $\mathsf{Eng}(\mathsf{boss}(a))$, $\mathsf{Mgr}(\mathsf{boss}(a))$ and $\neg\mathsf{Mgr}(a)$, finds every clause satisfied, and returns `OPEN`. At level 1 the six ground clauses go to `DPLL`, which propagates $\mathsf{Eng}(a)$, $\mathsf{Eng}(\mathsf{boss}(a))$ and $\mathsf{Mgr}(\mathsf{boss}(a))$, finds $\neg\mathsf{Mgr}(\mathsf{boss}(a)) \vee \neg\mathsf{Eng}(\mathsf{boss}(a))$ falsified, and returns `CLOSED` without a decision. The loop returns `UNSAT`: the rules entail the claim, by a single branch with no guess, like Harriet's first search.

With the weaker policy $\{C_1, C_3, C_4\}$ the loop never returns. Figure 7.4 sets out both runs level by level. At levels 0 and 1 forced moves alone satisfy the ground clauses. At level 2 there are five clauses and a decision, since $\mathsf{Mgr}(\mathsf{boss}(\mathsf{boss}(a)))$ is forced but nothing decides $\mathsf{Eng}(\mathsf{boss}(\mathsf{boss}(a)))$; and so on. Each level returns `OPEN` with a partial valuation that makes every $\mathsf{Eng}(t)$ and $\mathsf{Mgr}(t)$ it mentions true: the Herbrand structure in which everybody is an engineer and a manager, seen one level at a time. Harriet knows, from Example 7.4, that the weaker policy has a two-person counter-model. The procedure does not, and will not find out.

A warning about the inner answers. When $\texttt{DPLL}(\mathrm{Gr}_k(\mathcal{S}))$ returns `OPEN`, it has found a setting that obeys the instances at level k and nothing more; the instances one level up may still clash, as in Example 7.72. So `OPEN` at a level is not a verdict about $\mathcal{S}$, and `REFUTE` rightly ignores it. The question raised after Herbrand's theorem (Theorem 7.71), whether a finite amount of work can certify that a clause set is satisfiable, therefore has an answer for

this procedure: no finite level of instances does, and the loop does not try. Whether a cleverer procedure could is the question of Chapter 8.

Choosing the instances. The loop lists every instance at a level, and most of what it lists is useless. A clash between instances of two clauses needs an atomic sentence that one instance contains and the other contains negated. Before anything is listed, the two clauses carry literals such as

$$\mathsf{Mgr}(\mathsf{boss}(x)) \qquad \text{and} \qquad \neg\mathsf{Mgr}(y),$$

and the instances that can clash on them are those in which the two atomic formulas become the same, here by putting $\mathsf{boss}(x)$ for y. Finding such a substitution means solving an equation between them, written with $\doteq$ to keep it apart from the relation symbol $=$,

$$\mathsf{Mgr}(\mathsf{boss}(x)) \doteq \mathsf{Mgr}(y),$$

and that can be done by machine. A short procedure, *unification*, either finds the most general solution, of which every other solution is an instance, or shows that there is none, in which case the two literals never clash at any level. Adding, before grounding, the instances that unification picks out never changes the verdict of the loop and never costs a level, and it often lets the loop close much sooner. The optional Section 7.10 gives the procedure and proves these claims.

Looking for small structures. There is a second search, the one everybody runs by hand when looking for a small counter-model: use as few elements as possible, start with one, and give every symbol the laziest meaning that works. It too is a propositional search. "Is $\mathcal{S}$ satisfied by a structure with exactly n elements?" becomes a clause set in three steps.

1. Name the elements $1, \ldots, n$, and take a letter for each atomic sentence over the names and for each equation $f(\bar{\imath}) = j$, with clauses saying that each $f(\bar{\imath})$ has exactly one value.
2. Instantiate every clause of $\mathcal{S}$ at every choice of names, replacing each term $f(\bar{\imath})$ by a name j and adding the literal $\neg(f(\bar{\imath}) = j)$.
3. Run `DPLL`: a setting that comes back is an n-element structure that satisfies $\mathcal{S}$, and `CLOSED` says that none exists.

For instance, with $n = 2$, step 2 turns the clause $\neg P(x) \vee Q(f(x))$ at $x = 1$ into two clauses, one for each value $f(1)$ may take:

$$\neg P(1) \vee \neg(f(1) = 1) \vee Q(1) \qquad \text{and} \qquad \neg P(1) \vee \neg(f(1) = 2) \vee Q(2),$$

that is, if 1 is in P and $f(1)$ is j, then j is in Q; and the same again at $x = 2$. Run this for $n = 1, 2, 3, \ldots$ alongside `REFUTE`, and one of the two answers whenever $\mathcal{S}$ is unsatisfiable or satisfied by a finite structure. Exercise 7.11 carries the encoding out.

Example 7.77 (the company, cont'd: a company of one)**.** For the weaker policy the search for small structures answers already at $n = 1$: a company of one, an engineer and manager who is her own boss, obeys C_1, C_3 and C_4. So the two-person company of Example 7.4 was not the smallest counter-model.

Remark 7.78 (at work: SMT solvers and quantifiers). SMT solvers handle universal quantifiers much as the loop does when it chooses its instances. They keep a ground problem for their SAT engine, and from time to time add to it instances of the universal sentences. They choose the instances by heuristics rather than level by level. One adds an instance when the terms it needs turn up in the ground problem; another looks at the setting found so far, finds a universal sentence it breaks, and adds the instance that shows where. Like `REFUTE`, they may run for ever, and when they stop without a refutation they may answer "unknown" rather than produce a model.

A search that may not stop. Every DPLL run stops for one reason only: the switches are finitely many, and each branch ends when they run out. The first-order procedure has no such bound. It stops when instances clash and only then, and whether they ever will is exactly the question being asked. A cleverer order of instances would not help, for on some questions no instances ever clash, in any order, although a small counter-model exists. The clause set

$$R(f(x), x), \qquad \neg R(y, g(y))$$

is one. Its ground instances have the shapes

$$R(f(t), t) \qquad \text{and} \qquad \neg R(t', g(t')),$$

so a clash needs $f(t)$ to be t' and t to be $g(t')$, that is, t to be $g(f(t))$, which no term is. Yet the structure with two elements, each R-related to the other, with f sending each to the other and g sending each to itself, satisfies the set, and the search for small structures, run alongside, finds it at $n = 2$.

Worse, some satisfiable sets are satisfied only by infinite structures. The sentences

$$\forall x\, \exists y\, R(y, x), \qquad \forall x\, \neg R(x, x), \qquad \forall x\, \forall y\, \forall z\, (R(x, y) \wedge R(y, z) \to R(x, z))$$

are satisfied by the natural numbers with $>$ for R, and by no finite structure, since a finite chain of ever-greater elements must come back on itself and break the second sentence or the third. On such a set the loop never stops, the search for small structures never stops, and both are right not to (Examples 7.79 and 7.80 read both questions in the company).

Example 7.79 (the company, cont'd: a question that never closes)**.** Does "everyone has someone who earns more than them" entail "someone earns more than everyone"? The clause form of the question is $\mathsf{Richer}(f(x), x)$ together with $\neg\mathsf{Richer}(y, g(y))$, the second from the negated claim, $\forall y\, \exists x\, \neg\mathsf{Richer}(y, x)$: the clause set above, with Richer for R. So the loop runs for ever, and choosing the instances by unification adds nothing, since the two atomic formulas never unify. Yet a company of two in which each earns more than the other is a counter-model, which the search for small structures finds at $n = 2$, once $n = 1$ has closed.

Example 7.80 (the company, cont'd: only infinite companies)**.** The three constraints "everyone has someone who earns more", "nobody earns more than themselves" and "if x earns more than y and y more than z then x more than z" are the three sentences above, with Richer for R. No finite company obeys them all, and on them neither search ever stops.

So the first-order procedure is a semi-decision procedure, and this chapter cannot say whether that is the best possible. A reader who has watched the loop run for ever on a question with a two-person answer will suspect that a cleverer search would stop; Chapter 8 is about that suspicion. What this chapter can say is where the cost sits. The Herbrand universe grows at least exponentially with the level as soon as there are two function symbols, or one with two places, and equality multiplies the instances further. Unification tames the extra instances; nothing tames the growth of the universe.

7.9. **Prenex and Skolem Form** *(optional).* This section and the next are optional: the main line quotes some of their results but none of their proofs. This one opens the black box of Section 7.6, giving the algorithm behind the theorem on the clause form (Theorem 7.52) and proving the theorem. The algorithm is what a careful person does by hand, and the only real idea in it is the last step, naming the witnesses.

What makes a quantifier awkward? A search by hand applies each constraint to the things it has named, and a constraint of the form "for every x, ..." is ready for that: put a name in for x and read what results. A constraint with "some" in it is not. "Some x is ..." does not say which, and the search has to invent a name for the someone before it can go on. Nor is a constraint ready when its quantifiers are buried inside it, under an arrow or a negation. There it is not even clear whether a quantifier says "every" or "some": in

$$\neg\forall x\, \mathsf{Eng}(x),$$

the $\forall$ says that *someone* is not an engineer.

The section first brings buried quantifiers to the front, and then names the witnesses. Here $\neg$ is a connective in its own right, as in Section 7.3, and $\top$ abbreviates $\neg\bot$.

Negations first. As in Section 7.6, a literal is an atomic formula or the negation of one, and a formula is in *negation normal form* if it is built from literals, $\bot$ and $\top$ by $\wedge$, $\vee$, $\forall$ and $\exists$ alone. The translation nnf of Section 7.3 gets four more equations, two for the quantifiers and two for a negation in front of one:

$$\begin{aligned} \mathrm{nnf}(\forall x\, \psi) &\coloneqq \forall x\, \mathrm{nnf}(\psi), & \mathrm{nnf}(\neg\forall x\, \psi) &\coloneqq \exists x\, \mathrm{nnf}(\neg\psi), \\ \mathrm{nnf}(\exists x\, \psi) &\coloneqq \exists x\, \mathrm{nnf}(\psi), & \mathrm{nnf}(\neg\exists x\, \psi) &\coloneqq \forall x\, \mathrm{nnf}(\neg\psi). \end{aligned}$$

In words: the negation of "everyone is ..." is "someone is not ...", and the negation of "someone is ..." is "nobody is ...". These are De Morgan's laws for the quantifiers. The first equation of the definition now leaves every literal alone, atomic formulas and their negations included.

The lemma that negation normal form is faithful gains four cases, and the two with a negation in front of a quantifier are the ones of substance: they are the quantifier clauses of the definition of satisfaction, read against the new equations. Take the equation for $\neg\exists x\, \psi$, and let $\mathfrak{M}$ be a structure and s an assignment. The induction hypothesis holds under every assignment,

so under each $s[d/x]$, and

$$\begin{aligned}\mathfrak{M}, s \models \forall x\, \mathrm{nnf}(\neg\psi) \quad &\text{iff} \quad \mathfrak{M}, s[d/x] \models \mathrm{nnf}(\neg\psi) \text{ for every } d \\ &\text{iff} \quad \mathfrak{M}, s[d/x] \not\models \psi \text{ for every } d \\ &\text{iff} \quad \mathfrak{M}, s \not\models \exists x\, \psi \\ &\text{iff} \quad \mathfrak{M}, s \models \neg\exists x\, \psi,\end{aligned}$$

by the clause for $\forall$, the induction hypothesis, the clause for $\exists$ and the clause for $\neg$, in turn. The equation for $\neg\forall x\, \psi$ is the same with the quantifiers exchanged, and the two equations without a negation pass the claim straight down. So $\mathrm{nnf}(\varphi)$ is true in a structure under an assignment exactly when φ is.

Quantifiers to the front. Why bring the quantifiers to the front at all? The rest of this section replaces each $\exists y$ by a function of the universal variables that y depends on, and those are the universal variables whose quantifiers stand above $\exists y$. In a formula like

$$\forall x\, \mathsf{Eng}(x) \vee \exists y\, \forall z\, \mathsf{Richer}(y, z)$$

finding them takes some looking. In a formula whose quantifiers all stand at the front they are read off the row: they are the universal variables to the left of $\exists y$.

Definition 7.81 (prenex form)**.** A formula is in prenex form when all its quantifiers stand at the front.

Formally, a formula is in *prenex form* if it is

$$Q_1 x_1 \cdots Q_n x_n\, \nu$$

with each Q_i one of $\forall$ and $\exists$, and ν quantifier-free. The string $Q_1 x_1 \cdots Q_n x_n$ is its *prefix* and ν its *matrix*. The prefix may be empty.

For instance,

$$\forall x\, \exists y\, \mathsf{Richer}(y, x)$$

is in prenex form, with prefix $\forall x\, \exists y$ and matrix $\mathsf{Richer}(y, x)$, while

$$(\forall x\, \mathsf{Eng}(x)) \to \mathsf{Eng}(\mathsf{ceo})$$

is not, since its quantifier sits under an arrow.

In negation normal form every quantifier sits under conjunctions and disjunctions only, and it can be pulled through those, provided its variable has nothing to do with the other side.

Lemma 7.82 (pulling a quantifier out)**.** *Let x not occur free in ψ, let Q be either quantifier and $\circ$ either of $\wedge$ and $\vee$. Then, for every structure $\mathfrak{M}$ and assignment s,*

$$\begin{aligned}\mathfrak{M}, s \models (Qx\, \varphi) \circ \psi \quad &\text{iff} \quad \mathfrak{M}, s \models Qx\, (\varphi \circ \psi), \\ \mathfrak{M}, s \models \psi \circ (Qx\, \varphi) \quad &\text{iff} \quad \mathfrak{M}, s \models Qx\, (\psi \circ \varphi).\end{aligned}$$

Proof. Take Q to be $\forall$ and $\circ$ to be $\vee$; the other seven cases are alike, and three of them are Exercise 7.13. This is where the side condition is used: since x is not free in ψ, the relevance lemma gives

$$\mathfrak{M}, s[d/x] \models \psi \quad \text{iff} \quad \mathfrak{M}, s \models \psi \qquad \text{for every } d.$$

So $\mathfrak{M}, s \models \forall x\,(\varphi \vee \psi)$ holds exactly when, for every d, $\mathfrak{M}, s[d/x] \models \varphi$ or $\mathfrak{M}, s \models \psi$.

- *If* $\mathfrak{M}, s \models \psi$, this holds, and so does $\mathfrak{M}, s \models (\forall x\,\varphi) \vee \psi$.
- *If not,* it holds exactly when $\mathfrak{M}, s[d/x] \models \varphi$ for every d, which is $\mathfrak{M}, s \models \forall x\,\varphi$, and so again exactly when $\mathfrak{M}, s \models (\forall x\,\varphi) \vee \psi$.

Two of the other seven cases, $\forall$ with $\wedge$ and $\exists$ with $\vee$, also use that a domain is never empty: over an empty domain $(\forall x\,\varphi) \wedge \psi$ would say no more than ψ, while $\forall x\,(\varphi \wedge \psi)$ would be true. □

When x is free in ψ, the lemma does not apply. The remedy is to rename the bound variable first: by the substitution lemma, $\forall x\,\varphi$ and $\forall z\,\varphi[z/x]$ are true under the same assignments when z is a new variable, and likewise for $\exists$.

Example 7.83 (a quantifier pulled out carelessly)**.** Without the side condition, the pull-out goes wrong. Take the formula

$$(\forall x\,\mathsf{Eng}(x)) \vee \neg\mathsf{Eng}(x),$$

whose second x is free, in a structure in which not everyone is an engineer, and let that x stand for someone who is one. There the formula is false: its first disjunct fails because not everyone is an engineer, and its second because that person is one. Pulling the quantifier out regardless gives

$$\forall x\,(\mathsf{Eng}(x) \vee \neg\mathsf{Eng}(x)),$$

which is true everywhere: the quantifier has captured the free x, which no longer speaks of the engineer it stood for. Renaming the bound variable first, to z, gives

$$\forall z\,(\mathsf{Eng}(z) \vee \neg\mathsf{Eng}(x)),$$

which is false at that engineer, as it should be.

Pulling the quantifiers out one at a time, renaming where needed, puts every formula in negation normal form into prenex form. Written as a function, the equations for $\wedge$ and $\vee$ first put the two parts into prenex form and then rename them apart: each bound variable of one part that occurs in the other is replaced by a new variable. Write the renamed parts as $\mathbf{Q}\bar{x}\,\nu_1'$ and $\mathbf{Q}'\bar{y}\,\nu_2'$, with prefixes $\mathbf{Q}\bar{x}$ and $\mathbf{Q}'\bar{y}$; after the renaming, none of $\bar{x}$ occurs in the second and none of $\bar{y}$ in the first.

Definition 7.84 (prenex translation)**.** The prenex translation of a formula in negation normal form pulls its quantifiers to the front, renaming bound variables apart as it goes.

Formally, for ν in negation normal form, the formula $\mathrm{pnf}(\nu)$ is defined by recursion on the shape of ν, by the equations

$$\begin{aligned}
\mathrm{pnf}(\nu) &\coloneqq \nu \qquad \text{for } \nu \text{ a literal, } \bot \text{ or } \top,\\
\mathrm{pnf}(\nu_1 \wedge \nu_2) &\coloneqq \mathbf{Q}\bar{x}\,\mathbf{Q}'\bar{y}\,(\nu_1' \wedge \nu_2'),\\
\mathrm{pnf}(\nu_1 \vee \nu_2) &\coloneqq \mathbf{Q}\bar{x}\,\mathbf{Q}'\bar{y}\,(\nu_1' \vee \nu_2'),\\
\mathrm{pnf}(\forall x\,\nu_1) &\coloneqq \forall x\,\mathrm{pnf}(\nu_1),\\
\mathrm{pnf}(\exists x\,\nu_1) &\coloneqq \exists x\,\mathrm{pnf}(\nu_1),
\end{aligned}$$

where, in the second and third, $\mathbf{Q}\bar{x}\,\nu_1'$ and $\mathbf{Q}'\bar{y}\,\nu_2'$ are $\mathrm{pnf}(\nu_1)$ and $\mathrm{pnf}(\nu_2)$ with each bound variable of one that occurs in the other replaced by a new variable. The *prenex form* of an arbitrary formula φ is $\mathrm{pnf}(\mathrm{nnf}(\varphi))$.

In words: translate the parts, rename so that the two prefixes cannot interfere, and write one prefix after the other. Which prefix comes first is a choice, and the lemma allows either; Example 7.88 shows that the choice can matter.

Lemma 7.85 (the prenex translation is faithful)**.** *For every formula φ,* $\mathrm{pnf}(\mathrm{nnf}(\varphi))$ *is in prenex form, and for every structure $\mathfrak{M}$ and assignment s,*

$$\mathfrak{M}, s \models \mathrm{pnf}(\mathrm{nnf}(\varphi)) \quad \text{iff} \quad \mathfrak{M}, s \models \varphi.$$

Proof. By the faithfulness of negation normal form (Lemma 7.18) it is enough to show, for every formula ν in negation normal form, that $\mathrm{pnf}(\nu)$ is true in exactly the same structures under exactly the same assignments as ν. We do so by induction on the shape of ν.

- *The first equation* changes nothing.
- *The equation for $\wedge$.* The renaming leaves the truth values of the parts alone, by the substitution lemma. The pull-out lemma then moves the quantifiers to the front one at a time, first those of $\mathbf{Q}\bar{x}$ and then those of $\mathbf{Q}'\bar{y}$, and each move is permitted because the renaming keeps the variable moved out of the other side. The equation for $\vee$ is the same.
- *The two quantifier equations.* The induction hypothesis, which holds under every assignment, passes the claim straight down.

That the result is prenex is read off the definition. □

Example 7.86 (the company, cont'd: the policy and two salary rules in prenex form)**.** Harriet's two rules and the claim C have one quantifier each, at the front already, so only the arrows need to go. The claim is negated, as a search for a counter-model needs. The negation normal forms are

$$\begin{aligned}\mathrm{nnf}(\mathrm{P}_1) &= \forall x\,(\neg \mathsf{Eng}(x) \vee \mathsf{Mgr}(\mathsf{boss}(x))),\\ \mathrm{nnf}(\mathrm{P}_2) &= \forall x\,(\neg \mathsf{Mgr}(x) \vee \neg \mathsf{Eng}(x)),\\ \mathrm{nnf}(\neg \mathrm{C}) &= \exists x\,(\mathsf{Eng}(x) \wedge \mathsf{Eng}(\mathsf{boss}(x))),\end{aligned}$$

the last by the equation for $\neg\forall$, then the equation for $\neg(\psi \to \chi)$, then double negation. All three are in prenex form.

Two salary rules have their quantifiers in the way of each other: "someone earns more than everyone" and "everyone has someone who earns more than them" are

$$\exists y\,\forall x\,\mathsf{Richer}(y, x) \qquad \text{and} \qquad \forall x\,\exists y\,\mathsf{Richer}(y, x).$$

Does the first entail the second? A counter-model would be a model of the first together with the negation of the second, that is, of

$$\exists y\,\forall x\,\mathsf{Richer}(y, x) \wedge \neg\forall x\,\exists y\,\mathsf{Richer}(y, x).$$

Its negation normal form is

$$\exists y\,\forall x\,\mathsf{Richer}(y, x) \wedge \exists x\,\forall y\,\neg\mathsf{Richer}(y, x).$$

The two prefixes share their variables, so the second is renamed before the pull-out:

$$\exists y\, \forall x\, \exists u\, \forall v\, (\mathsf{Richer}(y, x) \wedge \neg \mathsf{Richer}(v, u)).$$

Naming the witnesses. Every sentence now has its quantifiers in a row at the front. The universal ones are ready for a search: "for every x, ..." can be tested at any name by putting the name in for x. The existential ones are not, since "for some y, ..." does not say which y.

In a prenex sentence, an existential quantifier says that for each choice of the universally quantified variables before it there is a witness. A search by hand gives the witness a name. When the witness depends on the earlier variables, a name is not enough, and a function of them is needed. In

$$\forall x\, \exists y\, \mathsf{Richer}(y, x)$$

the person who earns more than x depends on x, and we write $f(x)$ for whoever it is:

$$\forall x\, \mathsf{Richer}(f(x), x).$$

With two existentials, each gets its own symbol, applied to the universal variables to its left:

$$\forall x\, \exists y\, \forall z\, \exists w\, \nu \qquad \text{becomes} \qquad \forall x\, \forall z\, \nu[f(x)/y][g(x, z)/w].$$

Definition 7.87 (Skolem form)**.** The Skolem form of a prenex sentence replaces each existentially quantified variable by a new function symbol applied to the universally quantified variables that precede it.

Formally, for a prenex formula φ and a list $\bar{x} = x_1, \ldots, x_k$ of variables, the formula $\mathrm{sk}_{\bar{x}}(\varphi)$ is defined by recursion on the prefix of φ, by the equations

$$\begin{aligned} \mathrm{sk}_{\bar{x}}(\varphi) &\coloneqq \varphi \qquad \text{for } \varphi \text{ quantifier-free,} \\ \mathrm{sk}_{\bar{x}}(\forall y\, \psi) &\coloneqq \forall y\, \mathrm{sk}_{\bar{x},y}(\psi), \\ \mathrm{sk}_{\bar{x}}(\exists y\, \psi) &\coloneqq \mathrm{sk}_{\bar{x}}(\psi[f(\bar{x})/y]), \end{aligned}$$

where, in the last, f is a k-place function symbol not in the signature and not used before (a constant symbol when $k = 0$). The *Skolem form* of a prenex sentence φ, written φ^{s}, is $\mathrm{sk}_{()}(\varphi)$, with the empty list; the new function and constant symbols are its Skolem functions and Skolem constants.

In words: walk along the prefix, remembering the universal variables passed. Keep a universal quantifier. At an existential one, name the witness as a function of the universal variables so far and drop the quantifier. What results is a *universal* sentence,

$$\forall x_1 \cdots \forall x_n\, \nu \qquad \text{with } \nu \text{ quantifier-free,}$$

in a larger signature.

The substitution $\psi[f(\bar{x})/y]$ must not capture a variable, and it cannot as long as no variable is bound twice in the prefix, for then the variables $\bar{x}$, bound in front of $\exists y$, are bound nowhere inside ψ. A prefix such as $\forall x\, \exists y\, \forall x$ is renamed apart first, to $\forall x\, \exists y\, \forall z$, and from now on we take every prefix to bind distinct variables.

Example 7.88 (the company, cont'd: Skolem forms)**.** The Skolem form of nnf(¬C) from Example 7.86 is

$$\mathsf{Eng}(a) \wedge \mathsf{Eng}(\mathsf{boss}(a)),$$

with the Skolem constant a: the name Harriet's first search in Example 7.4 gave to the engineer who reports to an engineer. The Skolem forms of the two salary rules are

$$\forall x\, \mathsf{Richer}(a, x) \qquad \text{and} \qquad \forall x\, \mathsf{Richer}(f(x), x)\,:$$

the first names one person, the second a person for each x.

The Skolem form of the four-quantifier sentence of Example 7.86 is

$$\forall x\, \forall v\, (\mathsf{Richer}(a, x) \wedge \neg \mathsf{Richer}(v, g(x))),$$

where u, which came after $\forall x$, became $g(x)$. The order of the prefix changes the form. Had the other prefix come first, u would have become a constant and y a function of v; and if both existentials come out before either universal, as the pull-out lemma (Lemma 7.82) also allows, there are two constants and no function symbol at all, which gives the shortest Skolem terms.

The symbol must be new each time. Skolemise $\exists x\, \mathsf{Eng}(x)$ and $\exists x\, \mathsf{Mgr}(x)$ with the same constant a, and the results $\mathsf{Eng}(a)$ and $\mathsf{Mgr}(a)$ are unsatisfiable together with the negation of $\exists x\, (\mathsf{Eng}(x) \wedge \mathsf{Mgr}(x))$. The two sentences would seem to entail that someone is both an engineer and a manager, but a company with one engineer and one manager, different people, shows that they do not.

Is Skolemising safe? It changes the sentence, and the new sentence is not true in the same structures as the old. What it must keep is the verdict, and the evidence must come back with it. The next theorem says that both survive.

Theorem 7.89 (Skolem's theorem)**.** *Let φ be a prenex sentence and φ^{s} its Skolem form.*

(i) Every structure that satisfies φ^{s} satisfies φ.

(ii) Every structure that satisfies φ satisfies φ^{s}, once the Skolem functions and constants are given suitable meanings.

In particular, $\{\varphi\}$ and $\{\varphi^{\mathrm{s}}\}$ are equisatisfiable.

Proof. Each existential quantifier is removed by one step of the definition of the Skolem form, and it is enough to prove both parts for one step, that is, for

$$\varphi = \forall x_1 \cdots \forall x_k\, \exists y\, \psi \qquad \text{and} \qquad \varphi' = \forall x_1 \cdots \forall x_k\, \psi[f(\bar{x})/y],$$

where ψ may still have quantifiers. The theorem then follows by induction on the number of existential quantifiers in the prefix, since each step starts from a prenex sentence.

- *(i).* There is a derivation in NJ of φ from φ'. For $k = 1$ it has three steps:

$$\dfrac{\dfrac{\dfrac{\forall x\, \psi[f(x)/y]}{\psi[f(x)/y]}\,\forall\mathrm{E}}{\exists y\, \psi}\,\exists\mathrm{I}}{\forall x\, \exists y\, \psi}\,\forall\mathrm{I}$$

The side condition of (∀I) is met, since x is not free in the only open hypothesis, which is a sentence; and $f(x)$ is substitutable for y in ψ, since the prefix binds distinct variables and so x is bound nowhere inside ψ. For larger k, instantiate k times and generalise k times. By the soundness of NJ, every structure that makes φ' true makes φ true.

- *(ii).* Let $\mathfrak{M} \models \varphi$. For every k-tuple $\bar{d}$ of elements of $\mathfrak{M}$, the quantifier clauses give an element e with

$$\mathfrak{M}, s[\bar{d}/\bar{x}][e/y] \models \psi;$$

by the relevance lemma the assignment s does not matter, since the free variables of ψ are among $\bar{x}$ and y. Choose one such e for each $\bar{d}$, and let the new symbol f denote the function that sends $\bar{d}$ to it. Call the expanded structure $\mathfrak{M}'$. The other symbols mean in $\mathfrak{M}'$ what they meant in $\mathfrak{M}$, and f does not occur in ψ, so $\mathfrak{M}'$ makes ψ true under the same assignments as $\mathfrak{M}$. Under $s[\bar{d}/\bar{x}]$ the term $f(\bar{x})$ has the value chosen for $\bar{d}$, so by the substitution lemma

$$\mathfrak{M}', s[\bar{d}/\bar{x}] \models \psi[f(\bar{x})/y] \qquad \text{for every } \bar{d},$$

that is, $\mathfrak{M}' \models \varphi'$. And $\mathfrak{M}'$ still makes φ true.

Equisatisfiability follows from the two parts. □

In words: dropping an existential quantifier in favour of a name costs nothing, in either direction.

The two parts do different jobs, and the chapter needs both.

- *Part (i) brings the evidence back.* A structure that satisfies the Skolem form, with the new symbols forgotten, satisfies the original sentence. This is the way back that every rewriting must come with, and it is the three-line derivation in NJ: this chapter set out to avoid derivations, because they are hard to find, yet here one does the job.
- *Part (ii) makes refutations count.* If the Skolem form has no model, neither has the sentence, since a model of the sentence would expand to one of the Skolem form. This is the half a refutation relies on, and it needs a choice, one witness for each $\bar{d}$, which no derivation makes for us.[8]

Only universals left. After Skolem form a sentence is universal, $\forall \bar{x}\, \nu$ with ν quantifier-free, and the matrix ν can be turned into clauses by the clause form of Section 7.5, with the atomic formulas in the role of the letters. The universal quantifiers are then dropped and left understood, and the clauses are read as in Section 7.6.

Definition 7.90 (clause form of a sentence)**.** The clause form of a sentence is the clause form of the matrix of its Skolem form.

Formally, for a sentence φ, let $\forall \bar{x}\, \nu$ be the Skolem form of $\mathrm{pnf}(\mathrm{nnf}(\varphi))$; the *clause form* of φ is $\mathrm{cl}(\nu)$, with the atomic formulas of ν in the role of letters. The clause form $\mathrm{cl}(\Gamma)$ of a finite set Γ of sentences is the union of the clause forms of its members, with Skolem symbols new to the whole set.

[8] When the domain is infinite the choosing is an application of the axiom of choice.

In words: bring the negations down and the quantifiers to the front, name the witnesses, drop the universal quantifiers, and turn what is left into clauses.

This is the algorithm of the theorem on the clause form (Theorem 7.52), and the theorem can now be proved.

Proof of Theorem 7.52. The rewritings are taken one at a time.

- *Negation normal form and prenex form.* Both translations are faithful, so Γ is satisfied by the same structures as the set of prenex forms of its members.
- *Skolem form.* By Skolem's theorem (Theorem 7.89), applied to each member with its own new symbols, that set is equisatisfiable with the set of Skolem forms $\forall \bar{x}\, \nu$, and a structure that satisfies the Skolem forms satisfies the prenex forms.
- *Clause form.* It remains to see that $\mathfrak{M} \models \forall \bar{x}\, \nu$ exactly when $\mathfrak{M}$ satisfies every clause of $\mathrm{cl}(\nu)$. Fix an assignment s, and let v_s be the valuation that gives each atomic formula of ν its truth value in $\mathfrak{M}$ under s. Then

  $$\begin{aligned} \mathfrak{M}, s \models \nu \quad &\text{iff} \quad v_s \models \nu && \nu \text{ read as propositional,} \\ &\text{iff} \quad v_s \models \mathrm{cl}(\nu) && \text{the normal form (Proposition 7.21),} \end{aligned}$$

  and $v_s \models \mathrm{cl}(\nu)$ says that some literal of each clause of $\mathrm{cl}(\nu)$ is true in $\mathfrak{M}$ under s. The sentence $\forall \bar{x}\, \nu$ is true in $\mathfrak{M}$ exactly when ν is true under every assignment, and $\mathfrak{M}$ satisfies the clause set exactly when, under every assignment, some literal of each clause is true; so the two agree.

□

Remark 7.91 (at work: how provers clausify). The clause form of a sentence can blow up for the reason of Section 7.3, and provers avoid it by Tseitin's translation. A subformula ψ with free variables $x_1, \ldots, x_k$ has no single truth value, so it gets a new k-place relation symbol N rather than a letter, with the definition $\forall \bar{x}\, (N(\bar{x}) \leftrightarrow \psi)$ added and $N(\bar{x})$ put for ψ; when ψ occurs only positively, the half $N(\bar{x}) \rightarrow \psi$ is enough.[9] A disjunction of n conjunctions then gives $2n + 1$ clauses rather than 2^n. Provers also skip the prenex form and Skolemise straight from negation normal form, so that each Skolem function takes only the universal variables in whose scope its quantifier actually stands.

7.10. **Resolution and Unification** *(optional).* This optional section answers three questions the main line raised and left open. What is a solver's "no" worth: is a closed tree a proof? How large must such a proof be? And how can the first-order procedure choose its instances instead of listing them all? The main line quotes three of the answers, but none of their proofs: the long pigeonhole refutations (Section 7.5), unification (Section 7.8), and the first-order procedure with unification (Chapter 8).

Start with the first question. When the procedure closes a branch, it has found a clause falsified there, which says that the setting on the branch is

[9] The one-directional definitions are due to Plaisted and Greenbaum (1986); Nonnengart and Weidenbach's chapter in the *Handbook of Automated Reasoning* (2001) surveys how provers compute small clause forms.

forbidden. But that clause is not all the branch knows, nor the most useful thing it knows. Suppose a branch dies because $\neg p \vee \neg q \vee \neg r$ is falsified, with p, q and r all on, and r was not a choice: the clause $\neg p \vee \neg q \vee r$ forced it. What the branch has discovered is that p and q cannot both be on, and no single clause of the set says so (node 4 of Example 7.34 is such a branch).

This section gives the rule by which such discoveries are written down, and shows that a closed DPLL tree is a proof in that rule. So the "no" of a solver is a proof, which a small program can check.

Definition 7.92 (resolution)**.** Resolution combines two clauses that disagree about one switch into a clause that does not mention it.

Formally, let C and D be clauses and p a letter with $p \notin C$ and $\neg p \notin D$. The *resolvent* of the clauses $p \vee C$ and $\neg p \vee D$, that is, of $\{p\} \cup C$ and $\{\neg p\} \cup D$, on p is the clause $C \vee D$, that is, the union $C \cup D$; and the rule of *resolution* passes from two clauses to their resolvent:

$$\frac{p \vee C \qquad \neg p \vee D}{C \vee D}\ p$$

Two clauses are *resolvable* on p if one contains p and the other $\neg p$.

In words: if at least one of p and the literals of C holds, and at least one of $\neg p$ and the literals of D holds, then at least one of the literals of C and D holds, since p and $\neg p$ cannot both. When C and D are both empty the resolvent is the empty clause, and the rule has found that p and $\neg p$ were both demanded.

The resolvent of $\neg p \vee \neg q \vee r$ and $\neg p \vee \neg q \vee \neg r$ on r is $\neg p \vee \neg q$: the two clauses say what r must do when p and q are both on, and between them they say that p and q cannot both be on. The resolvent of $p \vee q$ and $\neg p \vee q$ on p is $q \vee q$, which is the clause q. Two clauses may be resolvable on more than one letter, as $p \vee q$ and $\neg p \vee \neg q$ are; the rule is applied on one letter at a time, and resolving on p gives $q \vee \neg q$, a clause that forbids nothing.

Lemma 7.93 (soundness of resolution)**.** *Every valuation that satisfies two clauses resolvable on a letter p satisfies their resolvent on p.*

Proof. Let v satisfy $p \vee C$ and $\neg p \vee D$, with $p \notin C$ and $\neg p \notin D$.

- *v makes p true.* Then v does not make $\neg p$ true, so to satisfy $\neg p \vee D$ it makes some literal of D true, which is a literal of $C \vee D$.
- *v makes p false.* Then to satisfy $p \vee C$ it makes some literal of C true, which is a literal of $C \vee D$.

In either case v satisfies $C \vee D$. □

So a resolvent of two clauses of $\mathcal{S}$ can be added to $\mathcal{S}$ without changing which valuations satisfy the set, and so can a resolvent of clauses already added. A clause reached in this way is a consequence of $\mathcal{S}$, and a sequence of such steps is a proof.

Definition 7.94 (resolution derivation)**.** A resolution derivation is a tree of clauses in which every step is a resolution.

Formally, for a clause set $\mathcal{S}$, a *resolution derivation from $\mathcal{S}$* is a finite tree of clauses, root at the bottom, in which

- each leaf is a clause of $\mathcal{S}$;
- each other node is the resolvent, on some letter, of its two children.

Its *conclusion* is the clause at the root, and its *size* is its number of resolution steps, that is, of its nodes that are not leaves. We write $\mathcal{S} \vdash_{\mathrm{Res}} C$ if there is a resolution derivation from $\mathcal{S}$ with conclusion C.

The tree is drawn as a derivation, with the two premisses of each step above the bar and the letter resolved on beside it. From a clause set $\mathcal{S}$ containing $\neg p \vee \neg q \vee r$, $\neg p \vee \neg q \vee \neg r$ and $\neg p \vee q$,

$$\dfrac{\dfrac{\neg p \vee \neg q \vee r \qquad \neg p \vee \neg q \vee \neg r}{\neg p \vee \neg q}\, r \qquad \neg p \vee q}{\neg p}\, q$$

is a derivation from $\mathcal{S}$ with conclusion $\neg p$ and size two. A clause of $\mathcal{S}$ may appear at several leaves, since a derivation may use it more than once.

Definition 7.95 (resolution refutation)**.** A refutation derives the empty clause, the clause that nothing satisfies.

Formally, a *resolution refutation* of a clause set $\mathcal{S}$ is a resolution derivation from $\mathcal{S}$ with conclusion $\square$.

So $\mathcal{S}$ has a resolution refutation exactly when $\mathcal{S} \vdash_{\mathrm{Res}} \square$.

Corollary 7.96 (refutations are sound)**.** *If $\mathcal{S} \vdash_{\mathrm{Res}} \square$ then $\mathcal{S}$ is unsatisfiable.*

Proof. By the lemma, applied at each step from the leaves down, a valuation satisfying $\mathcal{S}$ would satisfy every clause of the refutation; and no valuation satisfies $\square$. $\square$

So a resolution refutation is a certificate of unsatisfiability, checked one step at a time against the definition of a resolvent.

The definition of a resolution derivation makes it a tree, so a clause that is needed twice must be derived twice, once in each subtree. If instead a derived clause may be used as often as one likes, the derivation is a directed acyclic graph, a dag, rather than a tree. Such a derivation is called *general* or *dag-like*, and a derivation in the sense of that definition is called *tree-like* when the contrast matters. Every general derivation can be unfolded into a tree-like one by copying the shared subderivations, so the two prove the same clauses; the difference is in the size, and the end of this section returns to it.

A closed tree is a refutation. Remark 7.35 promised that a closed DPLL tree, read from the leaves up, is a resolution proof. Here is the promise kept: every closed DPLL tree can be turned into a tree-like resolution refutation of about the same size. The idea is to label every node of the tree with a clause that the branch to that node has falsified: at a leaf, the clause that closed it; at a decision, a resolvent of the labels of the two children; at a node whose one child was forced, a resolvent of the child's label with the clause that forced the child. The label at the top is then falsified by a branch with no literals on it, and such a clause is empty.

Theorem 7.97 (a closed DPLL tree is a tree-like resolution refutation)**.** *Let $\mathcal{T}$ be a closed DPLL tree for $\mathcal{S}$. Then $\mathcal{S} \vdash_{\mathrm{Res}} \square$ by a tree-like refutation of size at most the number of nodes of $\mathcal{T}$ below the root.*

Proof. For a node N of $\mathcal{T}$, let B_N be the branch from the root to N, and say that a clause is *falsified at* N if every literal of it is false under the partial valuation of B_N. We show, by induction on the height of N, that for every node N below the root there is a tree-like derivation from $\mathcal{S}$ whose conclusion is falsified at N and whose size is at most one less than the number of nodes of the subtree below and including N. The same three cases, applied once more to the root, then give the theorem. (If the root itself is a closed branch, a clause of $\mathcal{S}$ is falsified with no literal false, so $\square \in \mathcal{S}$ and there is nothing to prove.)

- *N is a leaf.* The branch B_N is closed, so some clause C of $\mathcal{S}$ is falsified on it, that is, at N. The one-node derivation with conclusion C has size 0.
- *N has one child N', added by unit.* Let ℓ be the label of N', forced by the clause $C \in \mathcal{S}$, so that every literal of C other than ℓ is false at N. By the induction hypothesis there is a derivation $\mathcal{D}'$ whose conclusion D' is falsified at N'. The literals false at N' are those false at N together with $\bar{\ell}$.
  - If $\bar{\ell} \notin D'$, then D' is already falsified at N, and $\mathcal{D}'$ will do.
  - If $\bar{\ell} \in D'$, resolve D' with C on the letter of ℓ. The resolvent is $(D' \setminus \{\bar{\ell}\}) \cup (C \setminus \{\ell\})$, every literal of which is false at N. The new derivation has one more step than $\mathcal{D}'$, and the subtree at N has one more node than the subtree at N'.
- *N has two children N_1 and N_2, added by a decision on p,* labelled p and $\neg p$. By the induction hypothesis there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ with conclusions D_1, falsified at N_1, and D_2, falsified at N_2. The literals false at N_1 are those false at N together with $\neg p$, and at N_2 those false at N together with p.
  - If $\neg p \notin D_1$, then D_1 is falsified at N, and $\mathcal{D}_1$ will do; likewise $\mathcal{D}_2$ if $p \notin D_2$.
  - Otherwise $\neg p \in D_1$ and $p \in D_2$, and the resolvent of D_2 and D_1 on p is $(D_2 \setminus \{p\}) \cup (D_1 \setminus \{\neg p\})$, every literal of which is false at N. The derivation that resolves $\mathcal{D}_2$ with $\mathcal{D}_1$ has one more step than the two together, and the subtree at N has one more node than the subtrees at N_1 and N_2 together, so the bound holds with room to spare.

Applied to the root, whose branch carries no literals, the decision case or the unit case gives a derivation of size at most the number of nodes below the root. Its conclusion has every literal false on a branch on which no literal is false, so the conclusion is $\square$. The derivation is tree-like because each step combines the derivations of two different subtrees of $\mathcal{T}$, or one of them with a clause of $\mathcal{S}$, and uses each once. $\square$

In words: label each dead branch with the constraint it broke, and resolve the labels upwards; what arrives at the root is the empty clause. A closed DPLL tree is a resolution refutation with the clauses left unwritten.

Example 7.98 (the platform, cont'd: the reason, written down)**.** Harriet labels the closed tree of Example 7.34. Node 4 is closed by C_5 and node 5 by C_2. Then upwards.

- Node 4 was forced by C_4, and its label $C_5 = \neg p \vee \neg q \vee \neg r$ contains $\neg r$, the complement of the forced literal; resolving on r gives $\neg p \vee \neg q$, the label of node 3.
- Node 3 was forced by $C_1 = \neg p \vee q$, and $\neg p \vee \neg q$ contains $\neg q$; resolving on q gives $\neg p$, the label of node 1.
- Node 5 was forced by $C_3 = p \vee q$, and its label $C_2 = \neg q \vee p$ contains $\neg q$; resolving on q gives p, the label of node 2.
- Nodes 1 and 2 are the decision on p; their labels $\neg p$ and p resolve to $\square$.

The refutation, drawn as a derivation:

$$\dfrac{\dfrac{\dfrac{\neg p \vee \neg q \vee r \qquad \neg p \vee \neg q \vee \neg r}{\neg p \vee \neg q}\, r \qquad \neg p \vee q}{\neg p}\, q \qquad \dfrac{p \vee q \qquad \neg q \vee p}{p}\, q}{\square}\, p$$

It has four steps, for a tree with five nodes below the root. The clause $\neg p \vee \neg q$ in the middle is the rule nobody wrote down: the two rules about `compression` say between them that `fast-sync` and `offline-mode` cannot both be on, and the first rule then says that `fast-sync` cannot be on at all. The tree knew this at node 3, and the derivation says it.

The converse holds too, and with no loss. A tree-like resolution refutation of $\mathcal{S}$ with m nodes can be read as a DPLL tree for $\mathcal{S}$ with at most m nodes below the root: the letter resolved on at each step is the letter decided at the corresponding fork. The two kinds of certificate are the same up to a change of notation.[10] So the smallest closed DPLL tree for $\mathcal{S}$ and the smallest tree-like refutation of $\mathcal{S}$ are of the same size up to a constant, and to ask how large a DPLL tree must be is to ask about resolution proofs. The rest of this section answers it for two families of clause sets.

Pigeonholes. Put $n + 1$ pigeons into n holes, and some hole holds two. As a clause set, with a letter p_{ij} for "pigeon i is in hole j": for each pigeon i a clause $p_{i1} \vee \cdots \vee p_{in}$, saying that it is in some hole, and for each hole j and each pair of pigeons $i < k$ a clause $\neg p_{ij} \vee \neg p_{kj}$, saying that they are not both in it. The set is unsatisfiable, for the reason just given, and a person proves it in a line.

Every resolution refutation of it, tree-like or not, has size at least c^n for some constant $c > 1$.[11] So every closed DPLL tree for it is exponential in n, whatever letters are decided on and in whatever order, and no branching heuristic can help. The counting argument that a person uses is not available to resolution, which has no counting: it can only combine clauses two at a time. The exercises ask the reader to watch a solver fail on these sets as the number of holes grows.

[10] Buss and Nordström (2021), in their section on DPLL, state both directions, with the small point that a tree-like refutation may resolve on the same letter twice along a path, and the corresponding part of the proof is first pruned away.

[11] Haken (1985).

Trees against graphs. A second result about resolution points at what the solvers add to the procedure. There are clause sets that have general refutations of size proportional to n and no tree-like refutation of size less than $2^{cn/\log n}$ for some $c > 0$. The sets themselves have a number of clauses proportional to n, each of at most four literals.[12]

They are the *pebbling* sets, built from a directed graph in which every vertex above the bottom rests on two vertices below it. Each vertex i has two letters p_i and q_i, and "i holds" means that one of the two is on. The clauses say that each vertex at the bottom holds, that a vertex holds whenever the two it rests on hold (four clauses of four literals, one for each choice of a true letter below), and that the vertex at the top does not.

Working from the bottom up, the short refutation derives the clause $p_i \vee q_i$ once for each vertex i, and uses it at each vertex that rests on i; a tree-like refutation has to derive it again for every use. No cleverer tree avoids the repetition. The proof goes through a game of pebbles played on the graph, which gives the sets their name, and is beyond this chapter. On these sets the procedure is exponential although a short proof exists, and the reason is in the theorem that a closed tree is a refutation: a DPLL tree is a tree, and a tree cannot share a subproof.

Solvers get round the tree by keeping what the labelling finds. A clause derived from a closed branch is a consequence of $\mathcal{S}$, by the soundness of resolution (Lemma 7.93), so it can be added to the clause set without changing which valuations satisfy it, and unit propagation then uses it on every branch that follows. These are the learned clauses of Remark 7.35. A proof that uses one is no longer a tree, since the clause is derived once and used wherever the search meets it.

Example 7.99 (the platform, cont'd: the same lemma, many times)**.** Harriet's rule file has the five clauses of Example 7.15, on three switches, and ninety-seven more switches, which occur in other rules and, for the sake of the example, force nothing. Suppose the procedure decides on those ninety-seven first. Under each of those 2^{97} settings it builds the two branches of Example 7.34 again, and the labelling of the conversion theorem derives the clause $\neg p \vee \neg q$, then $\neg p$ and p, from the same five rules: the same four steps of work, 2^{97} times over. The refutation it yields is the four-step one of Example 7.98, since at a decision on an irrelevant switch both children are labelled $\square$ and one copy is kept. A good order defuses this: decide p first, and the ninety-seven switches are never decided. Nobody knows the good order in advance, and no order defuses the pebbling sets.

Learning defuses it whatever the order. The first time the search sets p on, whatever the other switches say, the branch closes by propagation alone, and the labelling gives its p node the label $\neg p$. A learned clause with a single literal holds whatever has been decided, so the solver jumps back to the root, undoing every decision, and adds $\neg p$ there. At the root, propagation now forces $\neg p$, then C_3 forces q, and C_2 is falsified, with no decision anywhere above. The labelling gives this closed branch the label p, and p and the learned $\neg p$ resolve to $\square$. A falsified clause at the root, with nothing left to

[12] Ben-Sasson et al. (2004), improving an earlier separation by Bonet et al. (2000).

undo, is how the solver knows that the clause set is unsatisfiable: two closed branches, and the ninety-seven other switches never come into it again.

Every solver of the kind this chapter is about, which is most of the solvers in use today, learns at every closed branch. The clause it learns often has all but one of its literals false already at a node far above the leaf that produced it. Then the solver does not step back to the last decision. It jumps back to the first such node, undoing the irrelevant decisions in between, and there the new clause forces its last literal. The proofs such a solver produces are general resolution refutations, and by the separation above, learning is not a refinement of the procedure but a change of proof system.

This is conflict-driven clause learning, and the chapter stops at its door. What lies behind it is the subject of the standard survey (Marques-Silva et al., 2021), and of the rest of a course on solvers: which clause to learn when a branch closes, how far to jump back, when to restart the search, which learned clauses to forget, and how to propagate without reading every clause at every step.

Unification. The rest of the section proves the claims made under "Choosing the instances" in Section 7.8: it gives the procedure that finds the instances worth writing down, and shows that it misses none.

A clash needs an atomic sentence that one instance contains and another contains negated. Before any instance is written, two clauses carry literals such as $\mathsf{Mgr}(\mathsf{boss}(x))$ and $\neg\mathsf{Mgr}(x)$, and the question is what to put for the variables so that the two atomic formulas (*atoms*, for short) become the same. Since a clause speaks of everybody, its variables are its own, and we may first rename them so that the two clauses share none, *renaming them apart*: $\neg\mathsf{Mgr}(x)$ becomes $\neg\mathsf{Mgr}(y)$. Then the equation to solve is $\mathsf{Mgr}(\mathsf{boss}(x)) \doteq \mathsf{Mgr}(y)$, and the answer is to put $\mathsf{boss}(x)$ for y. Solving equations of this kind is called *unification*.

Renaming apart is not a formality. Without it, the clauses $P(x)$ and $\neg P(f(x))$ give the equation $P(x) \doteq P(f(x))$, which nothing solves, since $f(x)$ is always longer than x; yet their ground instances $P(f(c))$ and $\neg P(f(c))$ clash. Renamed apart, the second is $\neg P(f(y))$, and $P(x) \doteq P(f(y))$ is solved by putting $f(y)$ for x.

Definition 7.100 (equation)**.** An equation asks for two expressions to be made identical.

Formally, an *equation* is a pair of terms, or a pair of atomic formulas, written $u \doteq t$.

We write $\doteq$ rather than $=$, since $=$ may be a relation symbol of the signature, and an equation asks for something else: that two expressions become identical, symbol for symbol.

Definition 7.101 (unifier)**.** A unifier is a substitution that solves a set of equations.

Formally, let E be a finite set of equations. A *unifier* of E is a substitution σ such that $u\sigma$ and $t\sigma$ are the same expression for every equation $u \doteq t$ in E.

For instance, the equation $\mathsf{Richer}(x, \mathsf{boss}(x)) \doteq \mathsf{Richer}(\mathsf{boss}(y), z)$ has both of the substitutions

$$[\mathsf{boss}(y)/x,\ \mathsf{boss}(\mathsf{boss}(y))/z],$$
$$[\mathsf{boss}(\mathsf{ceo})/x,\ \mathsf{ceo}/y,\ \mathsf{boss}(\mathsf{boss}(\mathsf{ceo}))/z]$$

as unifiers. The first makes both sides $\mathsf{Richer}(\mathsf{boss}(y), \mathsf{boss}(\mathsf{boss}(y)))$, and the second makes both sides $\mathsf{Richer}(\mathsf{boss}(\mathsf{ceo}), \mathsf{boss}(\mathsf{boss}(\mathsf{ceo})))$. The equation $\mathsf{Richer}(x, x) \doteq \mathsf{Richer}(\mathsf{boss}(y), y)$ has no unifier at all, since x would have to be both y and $\mathsf{boss}(y)$, and no substitution makes y and $\mathsf{boss}(y)$ the same: the second always comes out longer.

The two unifiers of $\mathsf{Richer}(x, \mathsf{boss}(x)) \doteq \mathsf{Richer}(\mathsf{boss}(y), z)$ are not equally good. Applying the first and then $[\mathsf{ceo}/y]$ to any expression gives the same result as applying the second, so the second is the first followed by a further decision, about who y is, that nothing in the equation asked for. A unifier that makes no such decision is the one to compute.

Definition 7.102 (most general unifier)**.** A most general unifier solves the equations while fixing nothing it does not have to.

Formally, let E be a finite set of equations. A unifier σ of E is a *most general unifier* of E if for every unifier τ of E there is a substitution ρ with $e\tau = (e\sigma)\rho$ for every expression e.

In words: every other solution is the most general one followed by some further substitution, and we say that τ is σ *followed by* ρ. For that equation, the second unifier is the first followed by $[\mathsf{ceo}/y]$, and the first is most general: it puts for x and z only what the equation forces, and leaves y alone.

The procedure. Equations between terms are solved from the outside in. If both sides start with the same symbol, a substitution makes them identical exactly when it makes the parts identical, so match the parts. If they start with different symbols, nothing will ever make them equal. When one side is a bare variable, the equation says what that variable must be, so write it down and put it in everywhere else. The one exception is a variable that occurs inside the other side, as in $y \doteq \mathsf{boss}(y)$: nothing solves that, since the right-hand side always comes out longer. Figure 7.5 sets this out as six rules on a set of equations, each naming the equation it acts on and what replaces it. Here f and g stand for function, constant or relation symbols, $\bar{u}$ and $\bar{t}$ for their lists of arguments, x for a variable and t for a term or an atom; "x in t" means that x occurs in t. A constant is a symbol with no arguments, so (decompose) simply deletes $c \doteq c$, and (clash) fails on $c \doteq d$. The test that (occurs) makes is called the *occurs check*.

Definition 7.103 (solved form)**.** A set of equations is in solved form when it is a list of answers for some of the variables, one each, that can be read off as a substitution.

Formally, a finite set of equations is in *solved form* if it is $\{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$ with $x_1, \ldots, x_n$ distinct variables, none of which occurs in any of $t_1, \ldots, t_n$. Its *substitution* is $[t_1/x_1, \ldots, t_n/x_n]$.

In words: each equation has a variable on the left, and a variable that has been given an answer occurs nowhere else. The set of the two equations

Rule	Equation	Replaced by
(delete)	$t \doteq t$	nothing
(decompose)	$f(u_1, \ldots, u_n) \doteq f(t_1, \ldots, t_n)$	$u_1 \doteq t_1, \ldots, u_n \doteq t_n$
(clash)	$f(\bar{u}) \doteq g(\bar{t})$, with $f \neq g$	*fail*
(orient)	$t \doteq x$, with t not a variable	$x \doteq t$
(occurs)	$x \doteq t$, with x in t and $t \neq x$	*fail*
(eliminate)	$x \doteq t$, with x not in t but in another equation	$x \doteq t$, with t put for x in every other equation

FIGURE 7.5. The rules of unification.

$x \doteq \mathsf{boss}(y)$ and $z \doteq \mathsf{boss}(\mathsf{boss}(y))$ is in solved form, with substitution $[\mathsf{boss}(y)/x, \mathsf{boss}(\mathsf{boss}(y))/z]$; the set of $x \doteq \mathsf{boss}(y)$ and $y \doteq \mathsf{ceo}$ is not, since y has an answer and also occurs in the first equation.

Definition 7.104 (the unification procedure)**.** The unification procedure applies the rules of unification until one fails or none applies.

Formally, the *unification procedure* takes a finite set E of equations and applies the rules of Figure 7.5 to it, one at a time and in any order, until a rule fails or no rule applies. In the first case it *fails*; in the second it *succeeds*, and its *result* is the set of equations it then has.

Theorem 7.105 (unification)**.** *For every finite set E of equations, the unification procedure, run on E with the rules applied in any order, stops, and:*

(i) it fails exactly when E has no unifier;

(ii) otherwise its result is in solved form, and the substitution of the result is a most general unifier of E.

Proof sketch.

- *The four rules that do not fail keep the unifiers.* A substitution unifies the set after such a step exactly when it unifies the set before. For (decompose), a substitution makes $f(\bar{u})$ and $f(\bar{t})$ identical exactly when it makes each u_i and t_i identical. For (eliminate), the equation $x \doteq t$ stays in the set, and a substitution that makes x and t identical does the same to another equation whether or not t has been put for x in it. Deleting $t \doteq t$ and turning an equation round change nothing either.
- *The two rules that fail are right to.* No substitution makes expressions with different first symbols identical; and none makes x identical with a term $t \neq x$ that contains it, since $t\sigma$ would contain $x\sigma$ as a proper part and so be longer.
- *A solved form is its own most general unifier.* Let $\sigma = [t_1/x_1, \ldots, t_n/x_n]$. No x_i occurs in any t_j, so $x_i\sigma = t_i = t_i\sigma$, and σ is a unifier. Every unifier τ of the solved form has $x_i\tau = t_i\tau = (x_i\sigma)\tau$, and σ leaves every other variable alone, so $e\tau = (e\sigma)\tau$ for every e: the unifier τ is σ followed by τ itself.

- *The procedure stops.* Call a variable *solved* if it occurs exactly once in the set, on the left of an equation. Each step lowers, in dictionary order, the triple: the number of unsolved variables, which (eliminate) lowers and no rule raises; the size of the set in symbols, which (decompose) and (delete) lower; and the number of equations with a non-variable on the left, which (orient) lowers. Triples of natural numbers cannot go down for ever.

When no rule applies, every equation has a variable on the left (else (delete), (decompose), (clash) or (orient) would apply), that variable is not in its right-hand side (else (occurs), or (delete) if the right-hand side is the variable itself), and it is in no other equation (else (eliminate)): the set is in solved form.

Since every stage has exactly the unifiers of E, a failure means that E has none, and a solved form at the end gives a most general unifier of E. As the procedure always stops, (i) and (ii) follow. □

In words: take the equations apart from the outside in, and you either hit a contradiction or arrive at the most general solution, which every other solution refines.

Example 7.106 (the company, cont'd: two equations)**.** Harriet runs the procedure on the equation $\mathsf{Richer}(x, \mathsf{boss}(x)) \doteq \mathsf{Richer}(\mathsf{boss}(y), z)$:

$$\begin{aligned}
&\mathsf{Richer}(x, \mathsf{boss}(x)) \doteq \mathsf{Richer}(\mathsf{boss}(y), z) && \\
&x \doteq \mathsf{boss}(y), \quad \mathsf{boss}(x) \doteq z && \text{(decompose)} \\
&x \doteq \mathsf{boss}(y), \quad \mathsf{boss}(\mathsf{boss}(y)) \doteq z && \text{(eliminate } x\text{)} \\
&x \doteq \mathsf{boss}(y), \quad z \doteq \mathsf{boss}(\mathsf{boss}(y)) && \text{(orient)}
\end{aligned}$$

No rule applies, and the set is in solved form, with most general unifier

$$[\mathsf{boss}(y)/x,\ \mathsf{boss}(\mathsf{boss}(y))/z].$$

On the equation with no unifier,

$$\begin{aligned}
&\mathsf{Richer}(x, x) \doteq \mathsf{Richer}(\mathsf{boss}(y), y) && \\
&x \doteq \mathsf{boss}(y), \quad x \doteq y && \text{(decompose)} \\
&x \doteq \mathsf{boss}(y), \quad \mathsf{boss}(y) \doteq y && \text{(eliminate } x\text{)} \\
&x \doteq \mathsf{boss}(y), \quad y \doteq \mathsf{boss}(y) && \text{(orient)}
\end{aligned}$$

and (occurs) fails on the last equation: no substitution makes y and $\mathsf{boss}(y)$ the same term. In the company Thorne is her own boss, but unification looks only at terms, and $\mathsf{boss}(\mathsf{ceo})$ and ceo are different terms.

Instances that can clash. Now the use. Two clauses can clash only through a pair of literals, one from each, with opposite signs and the same atom once the variables are filled in; and unification finds the most general way of filling them in.

Lemma 7.107 (instances that can clash)**.** *Let C and D be clauses with no variable in common, $R(\bar{u})$ a literal of C and $\neg R(\bar{t})$ a literal of D. If some ground instances $C\rho$ and $D\rho'$ have $R(\bar{u})\rho = R(\bar{t})\rho'$, then the equation $R(\bar{u}) \doteq R(\bar{t})$ has a most general unifier σ, and $C\rho$ and $D\rho'$ are instances*

of $C\sigma$ and $D\sigma$ by one and the same substitution: $C\rho = (C\sigma)\pi$ and $D\rho' = (D\sigma)\pi$ for some π.

Proof. The substitution ρ puts terms for the variables of C, and ρ' for those of D. This is where the hypothesis is used: since C and D share no variable, ρ and ρ' never disagree, and together they form one substitution τ with $C\tau = C\rho$ and $D\tau = D\rho'$. It is a unifier of the equation, since $R(\bar{u})\tau = R(\bar{u})\rho = R(\bar{t})\rho' = R(\bar{t})\tau$. By the unification theorem (Theorem 7.105) the procedure finds a most general unifier σ, and τ is σ followed by some π: so $C\rho = C\tau = (C\sigma)\pi$, and likewise $D\rho' = (D\sigma)\pi$. □

In words: every clash between instances of two clauses is an instance of one clash, the one unification computes. So for each pair of clauses, renamed apart, and each pair of literals of opposite sign, the instances worth writing down are $C\sigma$ and $D\sigma$, when σ exists. When the procedure fails, the two literals can never clash at all, however far up the levels a search goes: a clash would give a unifier, and by the unification theorem (Theorem 7.105) there is none.

Example 7.108 (the company, cont'd: four instances, none listed)**.** Harriet looks for clashes among the four clauses of Example 7.53, with C_2 renamed to $\neg\mathsf{Mgr}(y) \vee \neg\mathsf{Eng}(y)$. Five pairs of literals have opposite signs and the same relation symbol, and three of them do the work.

- $\neg\mathsf{Eng}(x)$ in C_1 against $\mathsf{Eng}(a)$, the unit C_3: the unifier is $[a/x]$, and C_1 gives $\neg\mathsf{Eng}(a) \vee \mathsf{Mgr}(\mathsf{boss}(a))$.
- $\neg\mathsf{Eng}(y)$ in C_2 against $\mathsf{Eng}(\mathsf{boss}(a))$, the unit C_4: the unifier is $[\mathsf{boss}(a)/y]$, and C_2 gives $\neg\mathsf{Mgr}(\mathsf{boss}(a)) \vee \neg\mathsf{Eng}(\mathsf{boss}(a))$.
- $\mathsf{Mgr}(\mathsf{boss}(x))$ in C_1 against $\neg\mathsf{Mgr}(y)$ in C_2: the unifier is $[\mathsf{boss}(x)/y]$, and C_2 gives $\neg\mathsf{Mgr}(\mathsf{boss}(x)) \vee \neg\mathsf{Eng}(\mathsf{boss}(x))$, a clause with a variable, standing for all its instances at once. Its instance at a is the clause of the second item, and C_1 is left as it was.

The other two pairs, $\neg\mathsf{Eng}(x)$ in C_1 against C_4 and $\neg\mathsf{Eng}(y)$ in C_2 against C_3, give C_1 at $\mathsf{boss}(a)$ and C_2 at a, which the clash does not need. With the units C_3 and C_4, the first two instances are the four ground clauses of the clash of Example 7.72, found without listing anything.

The procedure with unification. `REFUTE` can now choose its instances instead of listing them all, and the lemma on instances that can clash (Lemma 7.107) says that it loses nothing by doing so. Take two clauses C and D, with D replaced by a copy whose variables are renamed, in some fixed way, apart from those of C. Take a literal of C and a literal of opposite sign in D whose atoms unify, and let σ be the most general unifier that the unification procedure computes. The instances worth writing down are $C\sigma$ and $D\sigma$. For a clause set $\mathcal{S}'$, write $\texttt{CLASHES}(\mathcal{S}')$ for the set of all such instances, for all pairs of clauses of $\mathcal{S}'$ (a clause may be paired with a renamed copy of itself) and all such pairs of literals; it is finite, since $\mathcal{S}'$ is. A procedure that adds them before it grounds is:

```
procedure REFUTE-U(S)                -- REFUTE, with the instances chosen by unificati
    S' := S
```

```
k := 0
loop:
    S' := S' + CLASHES(S')        -- the instances in which two clauses of S' can c
    if DPLL(GROUND(S', k)) is CLOSED(T): return UNSAT
    k := k + 1
```

The new clauses are instances of clauses of $\mathcal{S}$, so they say nothing that $\mathcal{S}$ did not already say; they only bring the useful instances down to a level the procedure reaches early.

Proposition 7.109 (the first-order procedure with unification)**.** *For every clause set $\mathcal{S}$, the call* `REFUTE-U`$(\mathcal{S})$

(i) returns `UNSAT` *if $\mathcal{S}$ is unsatisfiable, at a level no higher than the one at which* `REFUTE`$(\mathcal{S})$ *does;*

(ii) runs for ever if $\mathcal{S}$ is satisfiable.

Proof. First, each round stops: $\mathcal{S}'$ stays finite, unification stops, and `DPLL` stops. A most general unifier of two literals uses only the symbols in them, so every clause added to $\mathcal{S}'$ uses only the symbols of $\mathcal{S}$, and $\mathcal{S}'$ has the Herbrand universe of $\mathcal{S}$; here `GROUND`$(\mathcal{S}', k)$ is $\mathrm{Gr}_k(\mathcal{S}')$. Every clause added to $\mathcal{S}'$ is an instance of a clause already in $\mathcal{S}'$, so, by induction on the rounds, of a clause of $\mathcal{S}$; hence every clause of `GROUND`$(\mathcal{S}', k)$ is a ground instance of a clause of $\mathcal{S}$.

- *$\mathcal{S}$ is satisfiable.* Some Herbrand structure $\mathfrak{H}$ satisfies it, by the lemma that Herbrand models suffice (Lemma 7.69). Then $v_{\mathfrak{H}}$ satisfies $\mathrm{Gr}(\mathcal{S})$, and hence every clause set `GROUND`$(\mathcal{S}', k)$. By the soundness of DPLL no call of `DPLL` returns `CLOSED`, and `REFUTE-U` never stops.
- *$\mathcal{S}$ is unsatisfiable.* By the theorem on the first-order procedure, `REFUTE`$(\mathcal{S})$ closes at some level k, so $\mathrm{Gr}_k(\mathcal{S})$ is unsatisfiable. Since $\mathcal{S}'$ contains $\mathcal{S}$ and has the same Herbrand universe, `GROUND`$(\mathcal{S}', k)$ contains $\mathrm{Gr}_k(\mathcal{S})$, and a clause set that contains an unsatisfiable one is unsatisfiable. So by the correctness of DPLL (Theorem 7.44), `REFUTE-U`$(\mathcal{S})$ has closed already when it reaches level k, or closes then.

□

In words: choosing the instances by unification never costs a level and never changes the verdict; on a satisfiable set it runs for ever, as `REFUTE` does.

Example 7.110 (the company, cont'd: the policy, with unification)**.** Harriet runs `REFUTE-U` on the four clauses of the policy. In the first round `CLASHES` adds the five clauses of Example 7.108, one for each pair of literals found there. At level 0 their ground instances, with those of the policy, are the six clauses of Gr_1 in Example 7.67, and `DPLL` closes them by three forced moves, without a decision. Plain `REFUTE` needed level 1.

The finance director's question of Chapter 6, worked through by hand in Exercise 7.12, closes at level 0 too. A machine run counts 111 ground clauses there, where plain `REFUTE` needs level 1 and 262 clauses. And the weaker policy still never stops: unification only ever adds instances of the rules, and instances of rules that have a model cannot clash.

Remark 7.111 (unification at work). Unification is older than the solvers and more widely used. The type checker of a language such as ML or Haskell infers the type of a program by writing down equations between types and solving them by these rules. The occurs check is why it rejects $\lambda x.\, x\, x$: for $x\, x$ to be well typed, x would need a type α equal to $\alpha \to \beta$, which contains it. Prolog runs a program by unifying the goal with the heads of its clauses, and pattern matching in a functional language is the special case in which only one side has variables.

Resolution with variables. `REFUTE-U` finds the instances $C\sigma$ and $D\sigma$ in which two clauses can clash, and then grounds them for `DPLL`. Resolution can skip the grounding: it resolves $C\sigma$ and $D\sigma$ directly, on the literal that unification made complementary, and the resolvent, with its variables still in it, stands for all of its instances at once. From $\neg P(x) \vee Q(f(x))$ and $\neg Q(y) \vee \neg P(y)$, with the most general unifier $[f(x)/y]$, it derives $\neg P(x) \vee \neg P(f(x))$. Resolution on clauses with variables, with unification choosing the instance, is the proof system on which first-order theorem provers were built for a generation; the History says who built it. This chapter stays with the propositional core, which is where the solvers of today have their engine, and leaves the lifted rule to a later course.

History. The ideas of this chapter arrived in the opposite order from the one in which it told them: the first-order machinery came first, and the propositional procedure was built to serve it. The chain begins with Löwenheim, who showed in 1915 that a satisfiable first-order sentence has a countable model, and with Skolem, who simplified the proof in 1920 by reducing every sentence to a normal form whose existential quantifiers are witnessed by chosen functions, the form that bears his name. In 1928 Skolem gave, in all but name, the result that a universal sentence is satisfiable exactly when its instances at the closed terms are jointly satisfiable.[13] Herbrand's thesis of 1930 proved what is now called his theorem, in the form that a prenex existential sentence is valid exactly when some finite disjunction of its instances is a propositional tautology, which is the dual of the form in this chapter. His proof analysed derivations directly, in a Hilbert-style system, which has axioms where NK has rules, and it had a gap, found and repaired three decades later.[14] The route taken in this chapter, through Herbrand structures and compactness, is the modern one and owes more to Skolem than to Herbrand.

The procedures came thirty years later, when the first computers made it worth asking what a program could prove. Gilmore programmed the instance enumeration on an IBM 704 in 1960, testing each finite set of instances by multiplying it out into a disjunction of conjunctions, and the multiplying defeated the machine on small examples.[15] Davis and Putnam (1960) replaced the test. Their paper is titled *A computing procedure for*

[13] Skolem (1920), Skolem (1928); see van Heijenoort (1967) for both in translation.

[14] Herbrand (1930); the repair is due to Dreben, Andrews and Aanderaa, and Gödel had noticed the gap earlier without publishing (Buss, 1995).

[15] Gilmore (1960).

quantification theory. The method Skolemises, enumerates the quantifier-free instances "in lexicographic order", and tests the first n of them for propositional consistency with three rules on clause sets: the unit rule of this chapter, the pure-literal rule, and a rule that eliminates a letter by replacing every pair of clauses that disagree about it with their resolvent. They wrote that the approach "goes back to Herbrand", and reported that Gilmore's program had run for twenty-one minutes without result on a formula their procedure settles by hand.

The last rule can increase the number and the length of the clauses at each application. When Davis, Logemann and Loveland (1962) programmed the procedure for the IBM 704 at New York University, with its 32,768 words of core memory, they replaced that rule by the decision step of this chapter, the *splitting rule*, which costs no memory beyond the current branch. The initials of the four, DPLL, now name the second procedure, and "Davis–Putnam" is often used for either. Their program, like this chapter's first-order procedure, was a first-order prover with a propositional core.

The enumeration of instances was the weak point, and Prawitz had seen in 1960 that the instances worth trying could be found by solving equations between terms instead of listing them.[16] Robinson (1965) made that into the resolution principle: the elimination rule of the first paper, applied to clauses with variables, with a unification algorithm choosing the instance. His abstract describes the method as based on "the fundamental theorem of Herbrand". The rule-by-rule presentation of unification used in this chapter is that of Martelli and Montanari (1982). Resolution made the propositional core dispensable and became the engine of automated theorem proving for a generation, which is why DPLL was, for thirty years, the less famous of the two descendants of the 1960 paper. The other road, generating instances by unification and handing them to a propositional solver, as `REFUTE-U` does, was taken up again in the 2000s by the *instance-based* provers, of which iProver is the best known (Ganzinger and Korovin, 2003; Korovin, 2008). The theorem that a closed DPLL tree is a tree-like refutation is usually stated without attribution; a careful statement of both directions is in the *Handbook of Satisfiability*, in its chapter on proof complexity (Buss and Nordström, 2021).

The translation that makes clause sets a general-purpose input, a new letter for each subformula, is Tseitin's. He presented it at a Leningrad seminar in 1966 and published it two years later (Tseitin, 1968), in a paper that also gave the first lower bounds on the size of resolution proofs of a restricted kind. The general question of how large resolution proofs must be became pressing after Cook (1971) showed that SAT is NP-complete (Levin (1973) proved the corresponding result independently). A proof system in which every unsatisfiable set had a short refutation would show NP = coNP, that every problem whose "yes" answers can be checked quickly has "no" answers that can be checked quickly too, which is not expected. Haken (1985) showed that resolution is not such a system: the pigeonhole sets have no refutation smaller than exponential. The separation of tree-like from general resolution

[16] Prawitz (1960). Herbrand had described a procedure for solving such equations in his thesis; Wirth et al. (2009) print its original text.

was proved by Bonet, Esteban, Galesi and Johannsen (2000) and sharpened to the near-optimal bound quoted above by Ben-Sasson, Impagliazzo and Wigderson (2004).

DPLL's second life began in the 1990s, when the verification of hardware produced propositional instances with hundreds of thousands of letters, and solvers that learn clauses were developed to meet them. The survey of Marques-Silva et al. (2021) tells their story. Their proofs have become objects of study in their own right. A solver's "no" is now expected to come with a proof in a standard format, DRAT, a list of the clauses learned and deleted that an independent program can check (Wetzler et al., 2014). Heule, Kullmann and Marek (2016) settled the Boolean Pythagorean triples problem, whether the numbers $1, \ldots, 7825$ can be split into two sets neither of which contains a triple with $a^2 + b^2 = c^2$: the answer was no. The computation took two days on eight hundred processors, and the proof, nearly two hundred terabytes long, was checked by an independent program: nobody had to trust the solver.

The whole enterprise is classical. Passing from "no model of Γ makes φ false" to "every model of Γ makes φ true" is bivalence, and in derivations the same step is double negation, which NK derives from excluded middle and NJ cannot. NJ refutes $\neg(p \vee \neg p)$, deriving $\neg\neg(p \vee \neg p)$, but not $p \vee \neg p$ itself, and a search for a world in which $p \vee \neg p$ is false collapses at once. An intuitionist would say that the collapse shows only that $p \vee \neg p$ cannot be false, which is less than showing it true; in a logic where the two differ, the method of this chapter does not apply as it stands. Figure 7.6 collects the dates.

Summary. A sentence follows from a theory exactly when the theory together with its negation has no model. So a proof can be a failed search for a counter-model, and once the things to be talked about are named, the search is propositional. The chapter solved that problem first.

A valuation is grown one switch at a time. Splitting on a letter loses nothing, as long as both sides are tried; greedy setting, which cannot go back, can miss a solution. To know when to stop and when a switch is forced, the search needs constraints it can read at a glance, and conjunctive normal form provides them. It is reached by pushing negations to the letters and multiplying out, and it is drawn as a matrix, through which a satisfying valuation is a path without a connection. Under a partial valuation a clause is satisfied, falsified or a unit, and a unit leaves the other branch empty, so its literal is forced. Setting what is forced, deciding what is not, and giving up on a branch as soon as a clause is falsified is DPLL, which is sound, complete and always stops.

A program stores the constraints as a clause set and the valuation as a set of literals, and throws each switch by restriction: it deletes the clauses the switch satisfies and crosses its complement out of the rest. A finished branch is a setting that anyone can check by plugging it in, and a closed tree is a certificate that anyone can check node by node. Each closed branch also proves a clause that the constraints imply without stating it, and solvers learn it. The first certificate is always short and the second need not be: deciding satisfiability is NP-complete, and whether it can always be done

Who	When	What
Löwenheim; Skolem	1915; 1920	countable models; existentials as functions
Skolem	1928	satisfiability by instances at closed terms
Herbrand	1930	the fundamental theorem
Gilmore	1960	instances enumerated and multiplied out
Davis & Putnam	1960	instances tested on clause sets; the unit rule
Prawitz	1960	instances found by solving equations
Davis, Logemann & Loveland	1962	the splitting rule: DPLL
Robinson	1965	resolution, with unification
Tseitin	1968	a new letter for each subformula
Cook; Levin	1971; 1973	SAT is NP-complete
Martelli & Montanari	1982	unification as rules on equations
Haken	1985	pigeonholes need exponential proofs
Bonet, Esteban, Galesi, Johannsen	2000	tree-like resolution is weaker
Ben-Sasson, Impagliazzo, Wigderson	2004	the separation made near optimal
Wetzler, Heule & Hunt	2014	DRAT: checkable proofs
Heule, Kullmann & Marek	2016	Pythagorean triples: 200 terabytes

FIGURE 7.6. SAT solving: who did what.

quickly is the open question whether P = NP. SMT solvers push richer questions down to the same engine.

Then the chapter reduced the first-order problem to the simpler one. Once every "someone" has been given a name, by a Skolem function of the things it depends on, the search need look only at structures made of names, and such a structure is a setting of switches, one for each atomic sentence about the names. A set of constraints has a model exactly when its ground instances at the closed terms have one, and when it has none, finitely many instances already clash. Running DPLL level by level on the ground instances, the first-order procedure certifies every entailment from finitely many premisses that holds, and never answers for one that fails: a semi-decision procedure. On some questions it runs for ever while a two-person counter-model exists, and on some every counter-model is infinite. The finiteness is the finiteness of a derivation in NK.

The optional sections add two things. First, the clause form is computed by bringing the quantifiers to the front and naming the witnesses, and the half of Skolem's theorem that turns a model of the Skolem form back into a model of the sentence is an NJ derivation three lines long. Second, a closed tree is a tree-like resolution refutation, a proof in a system of its own, and that proof system has limits: resolution cannot refute the pigeonhole sets in less than exponential size, so neither can DPLL. And some sets have short general refutations and only exponential tree-like ones, because a tree cannot reuse a lemma. The solvers in use today learn the lemma and add it to the clause set. Unification, solving equations between terms from the outside in, finds the instances that can clash without listing the rest.

Hilbert's programme — status.

Item	Status after this chapter
1. Consistency	• **holds**, in the small: every procedure of this chapter is sound, and never refutes a satisfiable clause set. That is consistency for a method, not the consistency of mathematics that Hilbert wanted, which Chapter 8 takes up
2. Completeness	• **holds**: every unsatisfiable clause set has a closed DPLL tree, and so a resolution refutation. Every first-order entailment from finitely many premisses that holds is found by the first-order procedure
3. Decision procedure	○ **in part**: propositional logic is decided, and solvers built on DPLL settle instances with millions of switches. No method is known that is fast on every instance, and none is known to be impossible. First-order logic is half decided. An entailment that holds is certified in finite time, but one that fails may never be

• holds ○ open, or settled in part × fails

For propositional logic the scoreboard's third question, whether a search that must end settles every question, is answered in principle and open in practice. For first-order logic the chapter gives half an answer, and does not know whether the other half is missing from the method or from the world.

What remains. Every DPLL run stops, because there are finitely many switches: each branch ends when the switches run out. The first-order procedure stops only when the answer is "no world", and on a question with a two-person answer it runs for ever. That may be the method's fault. The procedures of this chapter were found by looking at where a method wasted its work, and each improvement was a real one. Could a clever enough search always stop on questions about every employee, every number? Or could we at least describe the numbers so completely that any question about them is settled by a search that must end?

Exercises. The exercises run from practice to proof.

- Exercises 7.1 to 7.3 are about satisfiability and the two normal forms;
- Exercises 7.4 to 7.9 are about DPLL;
- Exercises 7.10 to 7.12 are about Herbrand's theorem and the procedure for first-order logic;
- Exercises 7.13 to 7.20 go with the two optional sections, and are optional too.

Exercise 7.1. Show that two sentences φ and ψ are true in the same structures if and only if $(\varphi \wedge \neg\psi) \vee (\psi \wedge \neg\varphi)$ is unsatisfiable, so that equivalence, like entailment and validity, is a question of satisfiability.

Exercise 7.2. Turn each constraint into conjunctive normal form by the two steps, and beside each clause write the pattern it forbids.

(a) $(p \to q) \to r$;
(b) $\neg(p \vee (q \wedge \neg r))$;
(c) $(p \to q) \wedge (q \to p)$;
(d) "exactly one of p, q, r is on" (write the clauses directly).
(e) Why is the empty clause false under every valuation, and the conjunction of no clauses, $\top$, true under every valuation?

Exercise 7.3. This exercise works out Tseitin's translation, sketched in Remark 7.28. For a formula φ, give every compound subformula ψ a new letter a_ψ, and for a letter p let a_p be p. For each compound ψ, with a for a_ψ and b, c for the letters of its parts, the *defining clauses* of ψ are

- $\neg a \vee \neg b$ and $a \vee b$, if ψ is $\neg\psi_1$;
- $\neg a \vee b$, $\neg a \vee c$ and $a \vee \neg b \vee \neg c$, if ψ is $\psi_1 \wedge \psi_2$;
- $\neg a \vee b \vee c$, $a \vee \neg b$ and $a \vee \neg c$, if ψ is $\psi_1 \vee \psi_2$;
- $\neg a \vee \neg b \vee c$, $a \vee b$ and $a \vee \neg c$, if ψ is $\psi_1 \to \psi_2$;

and the unit clause $\neg a_\bot$ is added if $\bot$ occurs. The translation $\mathrm{ts}(\varphi)$ is the set of all these clauses together with the unit clause a_φ. Call a valuation *honest* for φ if it makes each a_ψ true exactly when it makes ψ true.

(a) Let φ be $(p \wedge q) \vee ((r \wedge s) \vee (t \wedge u))$. Compute $\mathrm{cnf}(\mathrm{nnf}(\varphi))$ and $\mathrm{ts}(\varphi)$, naming the three conjunctions b, c, d, the inner disjunction e and the whole formula a, and count the clauses of each.
(b) Give a valuation that satisfies φ but not $\mathrm{ts}(\varphi)$, so that the two are not equivalent.
(c) Show, by induction on φ, that a valuation satisfies all the defining clauses of φ exactly when it is honest for φ.
(d) Deduce that every valuation that satisfies φ agrees on the letters of φ with one that satisfies $\mathrm{ts}(\varphi)$, and that every valuation that satisfies $\mathrm{ts}(\varphi)$ satisfies φ. Which of the two is the way back, and why does the second make $\{\varphi\}$ and $\mathrm{ts}(\varphi)$ equisatisfiable together with the first?

Exercise 7.4. Run unit propagation on the clause set p; $\neg p \vee q$; $\neg q \vee \neg r \vee s$; $\neg p \vee r$; $\neg s \vee t$; $\neg t \vee \neg q$, writing down each forced literal and the clause that forced it, and say what the procedure returns.

Exercise 7.5. Run the procedure on the clause set $p \vee q$; $\neg p \vee r$; $\neg q \vee r$; $\neg r \vee s$; $\neg p \vee \neg s$, deciding on p with the "on" side first, and draw the tree. Read off the setting it returns and check it against the five clauses.

Exercise 7.6. Give a second proof of the soundness of DPLL, which follows one satisfying valuation down the tree instead of working up from the leaves. Let $v \models \mathcal{S}$, and call a branch B of a DPLL tree for $\mathcal{S}$ *faithful* to v if $u_B \preceq v$.

(a) Show that every DPLL tree for $\mathcal{S}$ has a faithful branch.

(b) Show that a faithful branch is open, and conclude that no DPLL tree for $\mathcal{S}$ is closed.

Hint: for (a), argue by induction on the construction of the tree, one case for each rule. The root alone is a faithful branch, since $\varnothing \preceq v$. At a *decide* step on a faithful branch, v makes p true or false; at a *unit* step, use part (iii) of the lemma on the status of a clause; a step on another branch leaves the faithful branch as it was.

Exercise 7.7. The first of the two DPLL papers had a second rule, which the trees of this chapter leave out. Let $\mathcal{S}$ be a clause set and u a partial valuation. A literal ℓ is *pure* under u if its letter is undecided by u and $\bar{\ell}$ belongs to no clause $C \in \mathcal{S}$ with $u \not\models C$. The *pure-literal rule* passes from u to $u + \ell$ without a decision.

(a) Show that if ℓ is pure under u, then u is alive for $\mathcal{S}$ if and only if $u + \ell$ is. (*Hint:* given a valuation that extends u and satisfies $\mathcal{S}$, change the value of one letter.)

(b) Let $\mathcal{S} = \{p \vee q\}$ and $u = \varnothing$. Show that p and q are both pure. Of the four valuations of p and q, how many satisfy $\mathcal{S}$, and how many satisfy $\mathcal{S} \cup \{p\}$?

(c) At $u = \varnothing$ the rule replaces the question about $\mathcal{S}$ by the same question about $\mathcal{S} \cup \{\ell\}$. Show that the two sets are equisatisfiable, but need not be equivalent or have the same number of satisfying valuations. What is the way back, in the sense of Section 7.1? Why must a program that counts the satisfying valuations of a clause set, rather than finding one, switch the rule off, and need it switch off unit propagation too?

Exercise 7.8. The order of the decisions can change the size of a DPLL tree enormously. For $k \geq 1$ let $\mathcal{S}_k$ be the clause set

$$p \vee t, \quad p \vee \neg t, \quad \neg p \vee s, \quad \neg p \vee \neg s, \quad q_1 \vee r_1, \quad \ldots, \quad q_k \vee r_k.$$

(a) Show that $\mathcal{S}_k$ is unsatisfiable by drawing a DPLL tree that decides p first. How many nodes does it have below the root?

(b) Now let the procedure decide $q_1, \ldots, q_k$ first, in that order and each on first, and only then p. Draw the tree for $k = 1$, and show that for every k it has 2^{k+1} closed branches and $7 \cdot 2^k - 3$ nodes below the root.

(c) Which letter does the rule of thumb "decide a letter that occurs in many short clauses" pick first?

Exercise 7.9. Encode "three pigeons in two holes" as a clause set, with p_{ij} for "pigeon i in hole j", and run the procedure, deciding on p_{11} first and otherwise propagating. Draw the tree and count its nodes.

Exercise 7.10. Let $\mathcal{S}$ consist of the clauses $P(a)$, $\neg P(x) \vee P(f(x))$ and $\neg P(f(f(f(a))))$.

(a) Write down H_0, H_1 and H_2, and the Herbrand base up to level 2.

(b) Give a Herbrand structure that satisfies the first two clauses, as a set of atomic sentences.

(c) Run `REFUTE` by hand: write Gr_0, Gr_1 and Gr_2 and say at which level `DPLL` returns `CLOSED`, and by which forced moves.

Exercise 7.11. Let a be a constant, f and g one-place function symbols, P and Q one-place relation symbols and R a two-place relation symbol. Encode the question whether the clause set

$$\neg P(x) \vee Q(f(x)), \qquad P(a), \qquad P(f(a))$$

has a model with one element as a propositional clause set, as the search for small structures of Section 7.8 describes, and run `DPLL` on it. Then do the same with two elements for the clause set $R(f(x), x)$, $\neg R(y, g(y))$, and read off the structure.

Exercise 7.12. Let Σ be the signature with a constant c, a one-place function symbol f, one-place relation symbols P and Q, and two-place relation symbols R and $=$. Do the two sentences

$$\varphi_1 : \ \forall x \, \neg R(x, x) \qquad \text{and} \qquad \varphi_2 : \ \forall x \, (x \neq c \to R(f(x), x)),$$

together with the equality axioms, entail $\forall x \, (f(x) = x \to x = c)$?

(a) Write the two sentences and the negated claim as clauses, with a Skolem constant b.

(b) Write the equality clauses for Σ: reflexivity, symmetry, transitivity, and the congruence clauses of Remark 7.54.

(c) Find five ground clauses, at level at most 1, that no valuation satisfies, and say which forced moves show it.

(d) *(Optional.)* The clause set of (a) and (b) has, at level 1, well over two hundred ground instances. Say in a sentence why unification would find the five of (c) without listing the rest.

Exercise 7.13. *(Optional.)* Prove three more cases of the pull-out lemma, for x not free in ψ: $(\exists x \, \varphi) \vee \psi$, $(\forall x \, \varphi) \wedge \psi$ and $\psi \wedge (\exists x \, \varphi)$. Then, for a one-place relation symbol P, show by a counterexample that $(\forall x \, P(x)) \to \psi$ and $\forall x \, (P(x) \to \psi)$ are *not* in general true in the same structures, and say which quantifier the arrow turns the first into.

Exercise 7.14. *(Optional.)* Let P and Q be one-place relation symbols, R a two-place relation symbol and h a one-place function symbol. Give the prenex form and the Skolem form of each sentence, and say what the Skolem symbols stand for.

(a) $\forall x \, \exists y \, (R(y, x) \wedge \exists z \, R(z, y))$;

(b) $\neg \exists x \, \forall y \, R(x, y)$;

(c) $\forall x \, (P(x) \to \exists y \, (Q(y) \wedge h(x) = y))$;

(d) $\exists x \, P(x) \wedge \exists x \, Q(x)$, Skolemised once correctly and once with the same constant for both, and a structure that satisfies the first result and not the second.

Exercise 7.15. *(Optional.)* Write out the NJ derivation of the sentence $\forall x\,\forall y\,\exists z\,\varphi$ from its Skolem form $\forall x\,\forall y\,\varphi[f(x,y)/z]$, and check the side conditions of the two ($\forall$I) steps.

Exercise 7.16. *(Optional.)* Run the unification procedure on each equation, writing the set of equations after every rule as in Example 7.106, and give the most general unifier or say which rule fails. Here a and b are constants, f and h function symbols and P a relation symbol.

(a) $P(h(x,a),y) \doteq P(h(b,z),z)$;
(b) $P(h(x,y),x) \doteq P(h(z,z),a)$;
(c) $P(x,h(x,y)) \doteq P(h(y,z),z)$;
(d) $P(f(x),a) \doteq P(h(y,a),a)$.

Exercise 7.17. *(Optional.)* Take the clause set of Exercise 7.10: $P(a)$, $\neg P(x) \vee P(f(x))$ and $\neg P(f(f(f(a))))$.

(a) Find the instances that `CLASHES` adds in its first round. (Pair the second clause with a renamed copy of itself too.)
(b) Show that `REFUTE-U` closes at level 0, and say which forced moves close it. Plain `REFUTE` needed level 2.

Exercise 7.18. *(Optional.)* Resolution is sound, and only just.

(a) Write out the proof that every valuation satisfying $p \vee C$ and $\neg p \vee D$ satisfies $C \vee D$, in its two cases.
(b) Show that the resolvent need not follow from either premiss alone: give a valuation that satisfies $p \vee q$ but not the resolvent q of $p \vee q$ and $\neg p \vee q$.
(c) The clauses $p \vee q$ and $\neg p \vee \neg q$ disagree about two letters. Show that "resolving on both at once", which would give $\square$, is not sound, and say what the two legitimate resolvents are.

Exercise 7.19. *(Optional.)* Let $\mathcal{S}$ be $p \vee q$; $p \vee \neg q$; $\neg p \vee r$; $\neg p \vee \neg r$. Draw the DPLL tree that decides on p, label its nodes as in the proof of the theorem that a closed tree is a refutation, and draw the resolution refutation that results.

Exercise 7.20. *(Optional.)* Implement `DPLL` in a language of your choice, with the branching heuristic "the first undecided letter", and then `REFUTE` on top of it, with clauses as lists of literals and terms as nested lists.

(a) Run `DPLL` on the pigeonhole sets with $n+1$ pigeons and n holes for $n = 2, 3, \ldots$, counting the nodes of the tree. Say what you expect to see and why, and stop when a run takes longer than a minute.
(b) Run `REFUTE` on the clause set

$$\neg P(x) \vee Q(f(x)), \quad \neg Q(x) \vee \neg P(x), \quad P(a), \quad P(f(a)),$$

with a a constant, f a one-place function symbol and P, Q one-place relation symbols, and then on the same set without its second clause, printing the level and the verdict at each level, and stop the second run by hand.

References.

Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, 2004.

Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM Journal on Computing*, 30(5):1462–1484, 2000.

Samuel R. Buss. On Herbrand's theorem. In *Logic and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 1995.

Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 233–350. IOS Press, Amsterdam, second edition, 2021.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–64, 2003.

Paul C. Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of Research and Development*, 4(1):28–35, 1960.

Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39: 297–308, 1985.

Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris, 1930. Printed as Prace Towarzystwa Naukowego Warszawskiego, Wydział III, no. 33, Warsaw, 1930. English translation in *Logical Writings*, ed. W. D. Goldfarb, Harvard University Press, 1971.

Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing – SAT 2016*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 133–182. IOS Press, Amsterdam, second edition, 2021.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM, 2001.

Dag Prawitz. An improved proof procedure. *Theoria*, 26(2):102–139, 1960.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

Thoralf Skolem. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen. *Videnskapsselskapets skrifter, I. Matematisk-naturvidenskabelig klasse*, 4:1–36, 1920. English translation in van Heijenoort (1967).

Thoralf Skolem. Über die mathematische Logik. *Norsk matematisk tidsskrift*, 10: 125–142, 1928. English translation in van Heijenoort (1967).

G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, volume 8 of *Zapiski Nauchnykh Seminarov LOMI*, pages 234–259. Nauka, Leningrad, 1968. In Russian. English translation in J. H. Siekmann and G. Wrightson (eds.), *Automation of Reasoning 2*, Springer, 1983, pp. 466–483.

Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.

Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.

Claus-Peter Wirth, Jörg Siekmann, Christoph Benzmüller, and Serge Autexier. Lectures on Jacques Herbrand as a logician. SEKI Report SR-2009-01, DFKI, 2009. arXiv:0902.4682.