

Ganesh Lalitha Gopalakrishnan

Computation Engineering

applied automata theory and logic

Springer

Berlin Heidelberg New York

Hong Kong London

Milan Paris Tokyo

Basic Notions in Logic including SAT

This chapter is on propositional logic, first-order logic, and modern Boolean satisfiability methods. In particular, we will prove the undecidability of the validity of first-order logic sentences. Boolean satisfiability is discussed in sufficient detail to ensure that the reader is well motivated to study the theory of NP-completeness in Chapter 19.

Mathematical logic is central to formalizing the notion of assertions and proofs. Classical logics include *propositional* (zeroth-order), *first-order*, and *higher-order* logics. Most logics separate the notion of *proof* and *truth*. A proof is a sequence of *theorems* where each theorem is either an *axiom* or is obtained from previous theorems by applying a *rule of inference*.

Informally speaking, it is intended that all theorems are “true.” This is formalized through the notion of a *meaning function* or *interpretation* which maps a *well-formed formula* (a syntactically correct formula, often abbreviated *wff*) to its truth value. If a *wff* is true under all interpretations, it is called a *tautology* (or equivalently, it is called *valid*).

Therefore, when we later mention the term, “the validity problem of first-order logic,” we refer to the problem of deciding whether a given first-order logic formula is valid (this problem is later shown to be only *semi-decidable*, or equivalently TR or RE).

A logical system consisting of a collection of axioms¹ and rules of inference is *sound* if every theorem is a tautology. A logical system is *complete* if every tautology is a theorem. A logical system has *independent* rules of inference if omitting any rule prevents certain theorems from being derived.

¹ Note that axioms can be regarded as rules of inference that have an empty premise.

One uses the terminology of “order” when talking about logics. Propositional (zeroth-order) logic includes Boolean variables that range over true and false (or 1 and 0), Boolean connectives, and Boolean rules of inference. Predicate (first-order) logic additionally allows the use of *individuals* such as integers and strings that constitute infinite sets, variables that are quantified over such sets, as well as predicate and function *constants*. Second (and higher) order logics allow quantifications to occur over function and predicate spaces also. For example, in Chapter 5, Section 5.3, we presented two principles of induction, namely *arithmetic* and *complete*; the formal statement of both these induction principles constitutes examples of higher order logic. There are also special classes of logics that vary according to the kinds of *models* (interpretations) possible: these include *temporal logic*, which is discussed in Chapters 21 and 22. In these logics, notions such as *validity* tend to acquire specialized connotations, such as validity under a certain *model*. In this chapter, however, we will stick to the classical view of these notions.

Section 18.1 discusses a *Hilbert style* axiomatization of propositional logic due to Church. Section 18.2 begins with an example involving “quacks and doctors,” presents examples of interpretations for formulas, and closes off with a proof for the fact that the validity problem of first-order logic is undecidable, but semi-decidable (or equivalently, RE or TR).

We then turn our attention to the topic of satisfiability in the setting of propositional logic in Section 18.3. We approach the subject based on notions (and notation) popular in hardware verification - following how “Boolean logic” is treated in those settings (a model theoretic approach). We first examine two normal forms, namely the *conjunctive* and the *disjunctive* normal forms, and discuss converting one to the other. We then examine related topics such as \neq -satisfiability, 2-satisfiability, and satisfiability-preserving transformations.

18.1 Axiomatization of Propositional Logic

We now present one axiomatization of propositional calculus following the Hilbert style. Let the syntax of well-formed formulas (wff) be as follows:

$$Fmla ::= Pvar \mid \neg Fmla \mid Fmla \Rightarrow Fmla \mid (Fmla).$$

In other words, a formula is a propositional variable, the negation of a propositional formula, an implication formula, or a parenthesized for-

mula. This grammar defines a complete set of formulas in the sense that all Boolean propositions can be expressed in it (prove this assertion). Following Church [17], a Hilbert calculus for propositional logic can be set up based on three axioms (A1-A3) and two rules of inference (R1-R2) shown below. Here, p and q stand for propositional formulas.

A1: $p \Rightarrow (q \Rightarrow p)$

A2: $(s \Rightarrow (p \Rightarrow q)) \Rightarrow (s \Rightarrow p) \Rightarrow (s \Rightarrow q)$

A3: $(\neg q \Rightarrow \neg p) \Rightarrow (p \Rightarrow q)$

R1 (Modus Ponens): If P is a theorem and $P \Rightarrow Q$ is a theorem, conclude that Q is a theorem.

Example: From p and $p \Rightarrow (p \Rightarrow q)$, infer $p \Rightarrow q$.

R2 (Substitution): The substitution of wffs for propositional variables in a theorem results in a theorem. A substitution is a “parallel assignment” in the sense that the newly introduced formulas themselves are not affected by the substitution (as would happen if, for instance, the substitutions are made serially).

Example: Substituting $(p \Rightarrow q)$ for p and $(r \Rightarrow p)$ for q in formula $p \Rightarrow q$, results in $(p \Rightarrow q) \Rightarrow (r \Rightarrow p)$. It is as if p and q are replaced by fresh and distinct variables first, which, in turn, are replaced by $(p \Rightarrow q)$ and $(r \Rightarrow p)$ respectively.

We do *not* perform the substitution of $r \Rightarrow p$ for q first, and then affect the p introduced in this process by the substitution of $(p \Rightarrow q)$ for p .

Given all this, a proof for a simple theorem such as $p \Rightarrow p$ can be carried out – but it can be quite involved:

P1: From A1, through substitution of $p \Rightarrow p$ for q , we obtain

$$p \Rightarrow ((p \Rightarrow p) \Rightarrow p).$$

P2: From A2, substituting p for s , $p \Rightarrow p$ for p , and p for q , we obtain

$$(p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow (p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p).$$

P3: Modus ponens between **P1** and **P2** yields

$$(p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p).$$

P4: From A1, substituting p for q , we obtain

$$p \Rightarrow (p \Rightarrow p).$$

Modus ponens between **P4** and **P3** results in $(p \Rightarrow p)$. □

It is straightforward to verify that the above axiomatization is sound. This is because the axioms are true, and every rule of inference is truth-preserving. The axiomatization is also complete, as can be shown via a proof by induction. The take away message from these discussions is that it pays to hone the axiomatization of a formal logic into something that is parsimonious as well as enjoys the attributes of being sound, complete, and independent, as explained earlier.

18.2 First-order Logic (FOL) and Validity

Below, we will introduce many notions of first-order logic intuitively, through examples. We refer the reader to one of many excellent books in first-order logic for details. One step called *skolemization* merits some explanation. Basically, skolemization finds a witness to model the existential variable. In general, from a formula of the form $\exists X.P(X)$, we can infer $P(c)$ where c is a constant in the domain. Likewise, from $P(c)$, we can infer $\exists X.P(X)$; in other words, for an unspecified constant c in the domain,

$$\exists X.P(X) \Leftrightarrow P(c).$$

We will use this equivalence in the following proofs. There is another use of skolemization illustrated by the following theorem (which we won't have occasion to use):

$$\forall X.\exists Y.P(X, Y) \Leftrightarrow P(X, f(X)).$$

Here, f is an unspecified (but fixed) function. This equivalence is valid because of two reasons:

- The right-hand side leaves X as a free variable, achieving the same effect as $\forall X$ goes, as far as validity goes (must be true for all X).
- The right-hand side employs $f(X)$ to model the fact that the selection of $\exists Y$ may depend on X .

18.2.1 A warm-up exercise

Suppose we are given the following proof challenge:

Some patients like every doctor. No patient likes a quack. Therefore, prove that no doctor is a quack.

It is best to approach problems such as this using predicates to model the various classes of individuals² instead of employing ‘arbitrary constants’ such as p and d to denote *patients*, *doctors*, etc. Introduce predicate p to carve out a subset of people (P) to be patients, and similarly d for doctors. Let l (for “likes”) be a binary relation over P . Our proof will consist of instantiation of formulas, followed by Modus ponens, and finally proof by contradiction.

A1: The statement, “Some patients like every doctor:”

$$\exists x \in P : (p(x) \wedge \forall y : (d(y) \Rightarrow l(x, y))).$$

A2: The statement, “No patient likes a quack:”

$$\forall x, y \in P : (p(x) \wedge q(y) \Rightarrow \neg l(x, y)).$$

Proof goal: “No doctor is a quack.”

$$\forall x \in P : (d(x) \Rightarrow \neg q(x)).$$

Negate the proof goal: $\exists x \in P : (d(x) \wedge q(x))$.

Skolemize negated proof goal: $(x_0 \in P \wedge d(x_0) \wedge q(x_0))$.

Skolemize A1: $c \in P \wedge (p(c) \wedge \forall y : (d(y) \Rightarrow l(c, y)))$ (we will suppress domain membership assertions such as $c \in P$ from now on).

Specialize: From A1, we specialize y to x_0 to get:

$$p(c) \wedge (d(x_0) \Rightarrow l(c, x_0)).$$

But since $d(x_0)$ is true in the negated proof goal, we get:

$$p(c) \wedge l(c, x_0) \text{ (more irrelevant facts suppressed).}$$

Since $q(x_0)$ is true in the negated proof goal, we also get:

$$p(c) \wedge l(c, x_0) \wedge q(x_0).$$

Use A2 and specialize x to c and y to x_0 to get $\neg l(c, x_0)$.

Contradiction: Since we have $l(c, x_0)$ and $\neg l(c, x_0)$, we get a contradiction.

18.2.2 Examples of interpretations

Manna’s book [78] provides many insightful examples, some of which are summarized below. We focus on the notion of interpretations, which, in case of first-order logic: (i) chooses domains for constants, predicates, and function symbols to range over, and (ii) *assigns* to them. Examples below will clarify.

² Pun intended; in FOL, members of domains other than Booleans are called individuals.

Example 1

Consider the formula

$$Fmla1 = \exists F.F(a) = b \\ \wedge (\forall x).[p(x) \Rightarrow F(x) = g(x, F(f(x)))]$$

We will now provide *three distinct* interpretations for it.

Interpretation 1.

$$D = \text{Nat} \\ a = 0 \\ b = 1 \\ f = \lambda x.(x = 0 \rightarrow 0, x - 1) \\ g = * \\ p = \lambda x.x > 0$$

Interpretation 2.

$$D = \Sigma^* \\ a = \varepsilon \\ b = \varepsilon \\ f = \lambda x.(tail(x)) \\ g(x,y) = \text{concat}(y, \text{head}(x)) \\ p = \lambda x.x \neq \varepsilon$$

Interpretation 3.

$$D = \text{Nat} \\ a = 0 \\ b = 1 \\ f = \lambda x.x \\ g(x,y) = y+1 \\ p = \lambda x.x > 0$$

It is clear that under Interpretation 1, *Fmla1* is true, because there indeed exists a function *F*, namely the factorial function, that makes the assertion true. It is also true under Interpretation 2, while it is false under Interpretation 3 (Exercise 18.4 asks for proofs). Hence, this formula is *not* valid — because it is not true under all interpretations.

Example 2

Consider the formula

$$\begin{aligned}
Fmla2 &= \forall P.P(a) \\
&\wedge (\forall x).[x \neq a] \wedge P(f(x)) \Rightarrow P(x) \\
&\Rightarrow (\forall x.P(x))
\end{aligned}$$

We can interpret this formula suitably to obtain the principle of induction over *many* domains: for example, *Nat*, *strings*, etc.

18.2.3 Validity of first-order logic is undecidable

Valid formulas are those that are true under *all* interpretations. For example,

$$\forall x.f(x) = g(x) \Rightarrow \exists a.f(a) = g(a).$$

Validity stems from the innate structure of the formula, as it must remain true under *every conceivable* interpretation. We will now summarize Floyd's proof (given in [78]) that the validity problem for first-order logic is undecidable. First, an abbreviation: for $\sigma_i \in \{0, 1\}$, use the abbreviation

$$f_{\sigma_1, \sigma_2, \dots, \sigma_n}(a) = f_{\sigma_n}(f_{\sigma_{n-1}}(\dots f_{\sigma_1}(a) \dots)).$$

The proof proceeds by building a FOL formula for a given "Post system" (an instance of Post's correspondence problem).

Given a Post system $S = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_n, \beta_n)\}$, $n \geq 1$ over $\Sigma = \{0, 1\}$, construct the wff W_S (we will refer to the two antecedents of W_S as A1 and A2, and its consequent as C1):

$$\begin{aligned}
&\bigwedge_{i=1}^n p(f_{\alpha_i}(a), f_{\beta_i}(a)) && \text{(A1)} \\
&\bigwedge \forall x \forall y [p(x, y) \Rightarrow \bigwedge_{i=1}^n p(f_{\alpha_i}(x), f_{\beta_i}(y))] && \text{(A2)} \\
&\Rightarrow \exists z p(z, z) && \text{(C1)}
\end{aligned}$$

We now prove that S has a solution iff W_S is valid.

Part 1. (W_S valid) \Rightarrow (S has a solution).

If valid, it is true for all interpretations. Pick the following interpretation:

$$\begin{aligned}
a &= \varepsilon \\
f_0(x) &= x0 \text{ (string 'x' and string '0' concatenated)} \\
f_1(x) &= x1 \text{ (similar to the above)} \\
p(x, y) &= \text{There exists a non-empty sequence } i_1 i_2 \dots i_m \text{ such that} \\
&\quad x = \alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_m} \text{ and } y = \beta_{i_1} \beta_{i_2} \dots \beta_{i_m}
\end{aligned}$$

Under this interpretation, parts A1 and A2 of W_S are true. Here is why:

- Under the above interpretation, $f_{\alpha_i}(a) = \varepsilon\alpha_i = \alpha_i$ and similarly $f_{\beta_i}(a) = \beta_i$.
- Thus A1 becomes $\bigwedge_{i=1}^n p(\alpha_i, \beta_i)$. Each conjunct in this formula is true by p 's interpretation; hence A1 is true.
- The part $[p(x, y) \Rightarrow \bigwedge_{i=1}^n p(f_{\alpha_i}(x), f_{\beta_i}(y))]$ reduces to the following claim: $p(x, y)$ is true means that x and y can be written in the form $x = \alpha_{i_1}\alpha_{i_2}\dots\alpha_{i_m}$ and $y = \beta_{i_1}\beta_{i_2}\dots\beta_{i_m}$; the consequent of this implication then says that we can append some α_i and the corresponding β_i to x and y , respectively. The consequent is also true by p 's interpretation. Thus A2 is also true.
- Since W_S is valid (true), C1 must also be true. C1 asserts that the Post system S has a solution, namely some string z that lends itself to being interpreted as some sequence $\alpha_{i_1}\alpha_{i_2}\dots\alpha_{i_m}$ as well as $\beta_{i_1}\beta_{i_2}\dots\beta_{i_m}$. That is,

$$\alpha_{i_1}\alpha_{i_2}\dots\alpha_{i_m} = z = \beta_{i_1}\beta_{i_2}\dots\beta_{i_m}.$$

Part 2. (W_S valid) \Leftrightarrow (S has a solution).

If S has a solution, let it be the sequence $i_1i_2\dots i_m$. In other words, $\alpha_{i_1}\alpha_{i_2}\dots\alpha_{i_m} = \beta_{i_1}\beta_{i_2}\dots\beta_{i_m} = \text{Soln}$. Now, in order to show that W_S is valid, we must show that for *every* interpretation it is true. We approach this goal by showing that under every interpretation where the antecedents of W_S , namely A1 and A2, are true, the consequent, namely C1, is also true (if any antecedent is false, W_S is true, so this case is not considered).

From A1, we conclude that

$$p(f_{\alpha_{i_1}}(a), f_{\beta_{i_1}}(a))$$

is true. Now using A2 as a rule of inference, we can conclude through Modus ponens, that

$$p(f_{\alpha_{i_2}}(f_{\alpha_{i_1}}(a)), f_{\beta_{i_2}}(f_{\beta_{i_1}}(a)))$$

is true. In other words,

$$p(f_{\alpha_{i_1}\alpha_{i_2}}(a), f_{\beta_{i_1}\beta_{i_2}}(a))$$

is true. We continue this way, applying the functions in the order dictated by the assumed solution for S ; in other words, we arrive at the assertion that the following is true (notice that the subscripts of f describe the order in which the solution to S considers the α 's and β 's):

$$p(f_{\alpha_{i_1}\alpha_{i_2}\dots\alpha_{i_m}}(a), f_{\beta_{i_1}\beta_{i_2}\dots\beta_{i_m}}(a)).$$

However, since

$$\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_m} = \beta_{i_1} \beta_{i_2} \dots \beta_{i_m} = \text{Soln},$$

we have essentially shown that

$$p(f_{\text{Soln}}(a), f_{\text{Soln}}(a)).$$

Now, $p(f_{\text{Soln}}(a), f_{\text{Soln}}(a))$ means that there exists a z such that $p(z, z)$, namely $z = f_{\text{Soln}}(a)$. \square

18.2.4 Valid FOL formulas are enumerable

It was proved by Gödel in his dissertation that first-order predicate calculus is complete. In other words, there are axiomatizations of first-order logic in which every valid formula has a proof. Hence, by enumerating proofs, one can enumerate all valid first-order logic formulas. Thus the set of valid FOL formulas is recursively enumerable (or is Turing recognizable).

18.3 Properties of Boolean Formulas

18.3.1 Boolean satisfiability: an overview

Research on Boolean satisfiability methods (SAT) is one of the “hottest” areas in formal methods, owing to the fact that BDDs are often known to become exponentially sized, as already hinted in Chapter 11. In [11], the idea of model checking without BDDs was introduced. This work also coincided with the arrival on the scene, of efficient Boolean satisfiability methods. These modern SAT solvers (e.g., [81, 90]) followed the basic “DPLL” procedure presented in [33, 32], but made considerable improvements, including intelligent methods for backtracking search over the space of satisfying assignments, and improving the efficiency of computer memory system (e.g., cache) utilization in carrying out these algorithms. Easily modifiable versions of these tools are now freely available (e.g., [87]). Boolean SAT methods are now able to often handle extremely large system models, establish correctness properties, and provide explanations when the properties fail. Andersson [7] presents many of the basic complexity results, including the inevitability of exponential blow up, even in the domain of SAT. We discuss a summary of these issues, and offer a glimpse at the underlying ideas behind modern SAT tools. Unfortunately, space does not permit our detailed presentation of the use of SAT techniques or some of their unusual applications in system verification (e.g., [46]). The reader may find many tutorials on the Internet or web sites such as [105].

18.3.2 Normal forms

We now discuss several properties of Boolean (or propositional) formulas to set the stage for discussions about the theory of NP-completeness. There are two commonly used normal forms for Boolean formulas: the conjunctive normal form (CNF) and the disjunctive normal form (DNF).³ Given a Boolean expression (function) over n variables $x_1 \dots x_n$, a sum-of-products (SOP) or disjunctive normal form (DNF) expression for it is one where products of literals are disjoined (OR-ed) – a literal being a variable or its negation. For example, $nand(x, y)$ is expressed in SOP (DNF) as

$$nand(x, y) = \neg x \neg y + \neg x y + x \neg y.$$

A POS (CNF) expression for $nand$ would be $(\neg x + \neg y)$. A systematic way to obtain DNF and CNF expressions from truth tables is illustrated below: basically, the DNF form for a Boolean function is obtained by disjoining (OR-ing) the min terms at which the function is a 1, while the CNF form for a Boolean function is obtained by conjoining (AND-ing) the max terms at which the function is a 0. All these are illustrated in Figure 18.1, and details may be found in any standard digital system design text book (e.g., [12]).

Row	x	y	nand(x,y)	Minterm	Maxterm
0	0	0	1	$m_0 = \neg x \neg y$	$M_0 = (x + y)$
1	0	1	1	$m_1 = \neg x y$	$M_1 = (x + \neg y)$
2	1	0	1	$m_2 = x \neg y$	$M_2 = (\neg x + y)$
3	1	1	0	$m_3 = x y$	$M_3 = (\neg x + \neg y)$

Fig. 18.1. Min/Max terms for $nand$, whose DNF form is $m_0 + m_1 + m_2$ and CNF form is M_3 (the only max term where the function is 0)

Here are important points pertaining to these normal forms:

- Conversion between the CNF and the DNF forms incurs an exponential cost. This is best illustrated by a conversion program included in Figures 18.2 and Figures 18.3.⁴ You are encouraged to run this code

³ Strictly speaking, we should call the normal forms being discussed here the *canonical sum-of-products form* and the *canonical product-of-sums form*, since each product term or sum term consists of n literals.

⁴ These programs *do not prove* that the conversion is exponential, but provide strong intuitions as to why.

under an Ocaml system. You will see that an exponential growth in formula size can occur, as demonstrated in the tests at the end of Figure 18.3. While these tests indicate the results on DNF to CNF conversion, the same complexity growth exists even for CNF to DNF conversion (Exercise 18.8 asks you to modify this program to one that obtains the DNF form of a given formula. The modifications are based on the duality between \vee and \wedge , and hence the complexity follows).

- CNF satisfiability will be shown to be NP-complete in the next chapter, whereas DNF satisfiability is linear-time (see exercise below). However, due to the exponential size blow up, one cannot “win” against NP-completeness by converting CNF satisfiability to DNF satisfiability.

18.3.3 Overview of direct DNF to CNF conversion

Consider Figure 18.3 in which some terms representing DNF formulas are given. In particular, note that formula `f5` is a 4-DNF formula with eight product terms. The expression `List.length (gencnf(f5) 1 []) .cnf` converts this formula to CNF, and measures its length, which is found to be 65536, showing the exponential growth in length. Here is a brief explanation of this code.⁵

In Figure 18.2, the types for literals, clauses, and CNF formulas are given. The Ocaml List of lists `[[1; 3; -2]; [1; -1]]` stands for the CNF formula $(x_1 \vee x_3 \vee \neg x_2) \wedge (x_1 \vee \neg x_1)$. The syntax of formulas, `fmla`, declares what formula terms are permissible. According to these conventions, *true* is represented by `[]`, *i.e.*, an empty list of clauses, while *false* is represented by `[[]]`, *i.e.*, one empty clause.

The program proceeds driven by pattern matching. Suppose `fmla` matches *true* (`Tt`); this results in an Ocaml record structure as shown in the following line:

```
Tt    -> {cnf=[]; nvgen=0; algen=[]}
```

Note that `[]` is how *true* is represented. This is a list of lists with the outer list empty. Since the outer list is a list of conjunctions, an empty outer list corresponds to the basis case of a list of conjunctions, namely *true*. Likewise, `[[]]` represents *false* because the inner list is empty, and its basis case (for a list of disjunctions) is *false* (`Ff`). We also record the number of new variables generated in `nvgen`, and an association list of

⁵ This code can convert arbitrary Boolean expressions to CNF - we choose to illustrate it on DNF.

```

type literal = int          (* 1, 3, -2, etc          *)
type clause = literal list (* [1; 3; -2] -> [ ] means TRUE and [ [ ] ] means FALSE *)
type cnffmla = clause list (* [[1; 3; -2]; [1; -1]] -> (x1 ∨ x3 ∨ ~x2) ∧ (x1 ∨ ~x1) *)

type fmla =
  Ff | Tt | Pv of string      | And of fmla * fmla | Or  of fmla * fmla
  | Eq of fmla * fmla | Imp of fmla * fmla | Not of fmla

type cnf_str = {cnf:cnffmla;          (* CNF formula as a list *)
               nvgen:int;           (* Variable allocation index *)
               algen: (string * int) list} (* Association list between variable-names
                                           and integers representing them *)

let rec gencnf(fmla)(next_var_int)(var_al) =
  match fmla with
  | Tt -> {cnf=[ ]; nvgen=0; algen=[]}
  | Ff -> {cnf=[[]]; nvgen=0; algen=[]}
  | Pv(s) ->
    if (List.mem_assoc s var_al)
    then {cnf = [ [ (List.assoc s var_al) ] ];
          nvgen = 0; algen = []}
    else {cnf = [ [ next_var_int ] ]; nvgen = 1; (* 1 new var generated *)
          algen = [(s,next_var_int)]}

  | And(q1,q2) ->
    let {cnf=cnf1; nvgen=nvgen1; algen=algen1}
      = gencnf(q1)(next_var_int)(var_al) in
    let {cnf=cnf2; nvgen=nvgen2; algen=algen2}
      = gencnf(q2)(next_var_int + nvgen1)(algen1 @ var_al) in
    {cnf=doAnd(cnf1)(cnf2); nvgen=nvgen1+nvgen2;
     algen=algen1 @ algen2}

  | Or(q1,q2) ->
    let {cnf=cnf1; nvgen=nvgen1; algen=algen1}
      = gencnf(q1)(next_var_int)(var_al) in
    let {cnf=cnf2; nvgen=nvgen2; algen=algen2}
      = gencnf(q2)(next_var_int + nvgen1)(algen1 @ var_al) in
    {cnf=doOr(cnf1)(cnf2); nvgen=nvgen1+nvgen2;
     algen=algen1 @ algen2}

  | Imp(q1,q2) -> gencnf(Or(Not(q1),q2))(next_var_int)(var_al)
  | Eq(q1,q2) -> gencnf(And(Or(Not(q1),q2), Or(Not(q2),q1)))(next_var_int)(var_al)
  | Not(Pv(s)) ->
    if (List.mem_assoc s var_al)
    then {cnf = [ [ (0-(List.assoc s var_al)) ] ];
          nvgen = 0; algen = []}
    else {cnf = [ [ (0-next_var_int) ] ]; nvgen = 1; (* 1 new var generated *)
          algen = [(s,next_var_int)]}

  | Not(q1) ->
    let {cnf=cnf1; nvgen=nvgen1; algen=algen1}
      = gencnf(q1)(next_var_int)(var_al) in
    {cnf=doNot(cnf1); nvgen=nvgen1;
     algen=algen1}

and
doAnd(cnf1)(cnf2) =
(match (cnf1,cnf2) with
 | ([],cnf2') -> cnf2'
 | (cnf1',[]) -> cnf1'
 | ([[]],_) -> [[]]
 | (_, [[]]) -> [[]]
 | _ -> List.append(cnf1)(cnf2) )

(* See PART-2 ... *)

```

Fig. 18.2. A CNF generator, Part-1 (continued in Figure 18.3)

```

and
doOr(cnf1)(cnf2) =
  (match (cnf1,cnf2) with
  | ([,_)      -> []
  | (_,[])     -> []
  | ([[ ]],cnf2') -> cnf2'
  | (cnf1',[[ ]]) -> cnf1'
  | ([c11],[c12]) -> [List.append(c11)(c12)]
  | (cnf1',[c12]) -> doOr([c12])(cnf1')
  | (cnf1', (c12::cls)) ->
    let door1 = doOr(cnf1')([c12]) in
    let door2 = doOr(cnf1')(cls) in
    doAnd(door1)(door2) )

and
doNot(cnf1) =
  (match cnf1 with
  | [] -> [[]]
  | [[]] -> []
  | (c1::cls) ->
    let compclause = comp_clause(c1) in
    let donot' = doNot(cls) in
    doOr(compclause)(donot' )

and
comp_clause(cclause) =
  (match clause with
  | [] -> []
  | (lit::lits) ->
    let cl = 0-lit in (* complement literal *)
    let cl_cnf = [[cl]] in (* turn literal into CNF *)
    let rest = comp_clause(lits) in
    doAnd(cl_cnf)(rest) )

;;
(* To run tests, load these definitions, and type, e.g., gencnf(f5) 1 [];;
let f1 = And(And(Pv "x", Pv "y"), And(Pv "z", Pv "w"));
let f2 = And(And(Pv "p", Pv "q"), And(Pv "r", Pv "s"));
let f3 = Or(f1,f2);
let f4 = Or(f3,f3);
let f5 = Or(f4,f4);

f5 is really (xyzw + pqrs + xyzw + pqrs + xyzw + pqrs + xyzw + pqrs)

List.length ( gencnf( f5 ) 1 [] ).cnf;;
- : int = 65536

```

Fig. 18.3. A CNF generator, Part-2 (continued from Figure 18.2)

variable names to their integer values in `algen`. These are respectively 0 and [] for the case of `Tt`. Formula `Ff` is handled similarly.

A propositional variable is converted by the code under `Pv(s)`. We look up `s` in the association list `var_al`, and if found, generate its CNF with respect to the looked-up result. Else, we generate a CNF clause containing `next_var_int`, the next variable to be allocated, and also return `nvgen=1` and `algen` suitably records the association between `s` and `next_var_int`.

The remaining cases with propositional connectives are handled as shown: for `And`, we recursively convert its arguments, and call `doAnd` (Figure 18.3). The first four cases of `doAnd` deal with one of the argu-

ments being *false* or *true*. The last case simply appends `cnf1` and `cnf2`, as they are already in CNF form. In case of `doOr`, we perform a more elaborate case analysis. The case

```
| ([c11],[c12]) -> [List.append(c11)(c12)]
```

corresponds to two CNF clauses; here, we simply append the list of disjuncts to form a longer list of disjuncts. Given the case

```
| (cnf1', [c12]) -> doOr([c12])(cnf1')
```

This helps bring the first argument towards a single clause, at which time we can apply the `append` rule described earlier.

Let us discuss the last case:

```
| (cnf1', (c12::cls)) ->
  let door1 = doOr(cnf1')([c12]) in
  let door2 = doOr(cnf1')(cls) in
  doAnd(door1)(door2)
```

We recursively OR `cnf1'` with the head of the second list, namely `c12`, and the tail of the list, namely `cls`, and call `doAnd` on the result.

These are the places which cause an exponential size growth of formulas.

18.3.4 CNF-conversion using gates

To contain the size explosion during the conversion of Boolean formulas to CNF or DNF, one can resort to a circuit-based representation of clauses. This keeps the sizes of formulas *linear*, but introduces a linear number of *new* intermediate variables. Theoretically, therefore, the exponential cost remains hidden. This is because during Boolean satisfiability, an exponential number of assignments can be sought for these new variables. In practice, however, we can often avoid suffering this cost.

We now illustrate the idea of CNF conversion using gates through an example. Consider converting the DNF expression `xyzw + pqrs` to CNF. We build a circuit net-list as follows, by introducing temporary nodes `t1` through `t7`:

```
t1 = and(x,y)   ; t2 = and(z,w)   ; t3 = and(p,q)   ; t4 = and(r,s)
t5 = and(t1,t2) ; t6 = and(t3,t4) ; t7 = or(t5,t6)
```

Now, we convert each gate to its own CNF representation. We now illustrate one conversion in detail:

- $t1 = x \wedge y$.

- Treat it as $t1 \Rightarrow (x \wedge y)$ and $(x \wedge y) \Rightarrow t1$.
- The former is equivalent to $(\neg t1) \vee (x \wedge y)$ which is equivalent to $(\neg t1 \vee x) \wedge (\neg t1 \vee y)$.
- The latter is equivalent to $(\neg x \vee \neg y \vee t1)$.
- Hence, $t1 = x \wedge y$ can be represented through *three* clauses.

The following ML functions accomplish the conversion of AND gates and OR gates into clausal form ($t1 = x \wedge y$ was an example of the former):

```
let gen_andG_clauses({gatetype = Andg; inputs = (i1,i2); output = output} as andG)
  = [ [-i1; -i2; output]; [-output; i1]; [-output; i2] ]

let gen_orG_clauses ({gatetype = Org; inputs = (i1,i2); output = output} as org)
  = [ [i1; i2; -output]; [output; -i1]; [output; -i2] ]
```

Using these ideas, the final result for the example $xyzw + pqrs$ is the following 15 variables (8 variables were in the original expression, and seven more – namely, $t1$ through $t7$ – were introduced) and 22 clauses (7 gates, each represented using 3 clauses each; the last clause is the *unit* clause $t7$ — encoded as variable number 15).

```
x=1, y=2, z=3, w=4, p=5, q=6, r=7, s=8, t1=9, t2=10,
t3=11, t4=12, t5=13, t6=14, t7=15
```

```
[ [-1; -2; 9] ; [-9; 1] ; [-9; 2] ] t1 = and(x,y)
[ [-3; -4; 10] ; [-10; 3] ; [-10; 4] ] t2 = and(z,w)
[ [-5; -6; 11] ; [-11; 5] ; [-11; 6] ] t3 = and(p,q)
[ [-7; -8; 12] ; [-12; 7] ; [-12; 8] ] t4 = and(r,s)
[ [-9; -10; 13] ; [-13; 9] ; [-13; 10] ] t5 = and(t1,t2)
[ [-11; -12; 14] ; [-14; 11] ; [-14; 12] ] t6 = and(t3,t4)
[ [13; 14; -15] ; [15; -13] ; [15; -14] ] t7 = or(t5,t6)
[ 15 ] t7
```

18.3.5 DIMACS file encoding

At this point, having generated the gate net-list and their clauses, we now need to generate a file format representing the conjunction of these clauses. The standard format employed is the so called DIMACS format, where the literals in each clause are listed on separate lines, terminated by 0.

Given such a DIMACS file, one can feed the file to a Boolean satisfiability solver. Three examples of SAT solvers are [87, 81, 90], with reference [105] listing some of the latest news in the area of Boolean satisfiability.

```

p cnf 15 22 # Problem CNF format : Nvars and Nclauses.
-1 -2 9 0 # Each line lists the literals of each clause.
-9 1 0 # All lines end with a 0.
-9 2 0
-3 -4 10 0
-10 3 0
-10 4 0
-5 -6 11 0
-11 5 0
-11 6 0
-7 -8 12 0
-12 7 0
-12 8 0
-9 -10 13 0
-13 9 0
-13 10 0
-11 -12 14 0
-14 11 0
-14 12 0
13 14 -15 0
15 -13 0
15 -14 0
15 0

```

Type command ‘‘zchaff CNF-File’’ - here the file is "cnf"

The result of feeding the above file to zchaff is shown below:

```

[ganesh@localhost CH19]$ zchaff cnf
Z-Chaff Version: ZChaff 2003.11.04
Solving cnf .....
22 Clauses are true, Verify Solution successful.
Instance satisfiable

1 2 3 4 -5 -6 -7 -8 9 10 -11 -12 13 -14 15
Max Decision Level 7
Num. of Decisions 8
Num. of Variables 15
Original Num Clauses 22
Original Num Literals 50
Added Conflict Clauses 0
Added Conflict Literals 0
Deleted Unrelevant clause 0
Deleted Unrelevant literals 0
Number of Implication 15
Total Run Time 0

```

SAT

The SAT solver Zchaff picked the assignment given by the line

```
1 2 3 4 -5 -6 -7 -8 9 10 -11 -12 13 -14 15
```

which stands for the Boolean assignment

```
x=1, y=1, z=1, w=1, p=0, q=0, r=0, s=0, t1=1, t2=1,
t3=0, t4=0, t5=1, t6=0, t7=1
```

Suppose we force the assignment of $x=0$ by adding the clause $\neg x$: the assignment then becomes the one shown below, where p, q, r, s are set to 1:

```
-1 -2 -3 -4 5 6 7 8 -9 -10 11 12 -13 14 15
```

18.3.6 Unsatisfiable CNF instances

Consider the unsatisfiable CNF formula represented by the DIMACS file in Figure 18.4: An important theorem pertaining to unsatisfiable

```
p cnf 4 16
1 2 3 4 0
1 2 3 -4 0
1 2 -3 4 0
1 2 -3 -4 0
1 -2 3 4 0
1 -2 3 -4 0
1 -2 -3 4 0
1 -2 -3 -4 0
-1 2 3 4 0
-1 2 3 -4 0
-1 2 -3 4 0
-1 2 -3 -4 0
-1 -2 3 4 0
-1 -2 3 -4 0
-1 -2 -3 4 0
-1 -2 -3 -4 0
```

Fig. 18.4. An unsat CNF instance

CNF formulas is the following:

Theorem 18.1. In any unsatisfiable CNF formula, for any assignment, there will be *one clause with all its variables false and another with all its variables true*.

Proof: We can prove this fact by contradiction, as follows. Suppose $Fmla$ is an unsatisfiable CNF formula and there is an assignment σ under which one of the following cases arise (both violating the ‘one

clause with all its variables false and another with all its variables true' requirement):

- There are no clauses with all its literals assuming value true. In this case, we can complement the assignment σ , thus achieving a situation where no clause will have all *false* values.
- There are no clauses with all its literals assuming value false. In this case, $Fmla$ is satisfiable.

If we feed the file in Figure 18.4 to a SAT solver, it will conclude that the formula is indeed unsatisfiable. In modern SAT solvers, it is possible to find out the *root cause* for unsatisfiability. In particular, if the clauses of Figure 18.4 were embedded amidst many other clauses, upon finding the system to be unsatisfiable, a modern SAT solving framework is capable of extracting the clauses of Figure 18.4 as the root cause. This capability is being used during formal verification to provide explanations for failed proofs (e.g., [91, 109]).

18.3.7 3-CNF, \neq -satisfiability, and general CNF

A 3-CNF formula is a CNF formula in which every clause has three literals. For example,

$$(a \vee a \vee \neg b) \wedge (c \vee \neg d \vee e)$$

is a 3-CNF formula. As can be seen, there is no requirement on what variables the clauses might involve. Many proofs are rendered easier by standardizing on the 3-CNF form.

In many proofs, it will be handy to restrict the satisfying assignments allowed. One useful form of this restriction is the \neq -satisfiability restriction. Given a 3-CNF formula, a \neq -assignment is one under which every clause has two literals with unequal truth values. Given a set of N clauses where the i th clause is $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, suppose we transform each such clause into two clauses by introducing a new variable z_i per original clause, and a single variable b for the whole translation:

$$c_{i1} = (y_{i1} \vee y_{i2} \vee z_i) \text{ and } c_{i2} = (\neg z_i \vee y_{i3} \vee b).$$

We can show that for any given formula $Fmla = \bigwedge_{i \in N} c_i$, the above described transformation is a polynomial-time mapping reduction \leq_P from 3-SAT to \neq -sat.

Finally, a general CNF formula is one where the number of literals in each clause can vary (a special case where this number is ≤ 2 is discussed in Section 18.3.8). A conversion from general CNF to 3-CNF

that does not increase the length of the formula beyond a polynomial factor and takes no more than polynomial-time is now described (for each clause with less than 3 literals, replicating some literal can increase its size to 3; *e.g.*, $(a \vee b) = (a \vee b \vee b)$). Consider a clause with l literals:

$$a_1 \vee a_2 \vee \dots \vee a_l.$$

This clause can be rewritten into one with $l - 2$ clauses:

$$(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee z_3) \dots (\neg z_{l-3} \vee a_{l-1} \vee a_l).$$

Applying these transformations to a general CNF formula results in one that is equivalent *as far as satisfiability goes*. This is known as *equi-satisfiable*.

18.3.8 2-CNF satisfiability

A 2-CNF formula is one where each clause has two literals. One can show that the satisfiability of 2-CNF is polynomial-time, as follows:

- Consider one of the clauses $l_1 \vee l_2$. This can be written as two implications, $(\neg l_1 \Rightarrow l_2)$ and $(\neg l_2 \Rightarrow l_1)$.
- The complete algorithm is to process each clause of a 2-CNF formula into such a pair of implications. Now, viewing each such implication as a graph edge, we can connect its source to its destination; for example, in $(\neg l_1 \Rightarrow l_2)$, connect node $\neg l_1$ to node l_2 . In any situation, if a double negation is introduced, as in $\neg\neg a$, we must simplify it to a before proceeding.
- Finally, we connect pairs of edges such that the destination vertex of one edge is the source vertex of the other. For example, if $(\neg l_1 \Rightarrow l_2)$ is an edge, and $(l_2 \Rightarrow l_3)$ is another edge, connect these edges at the common point l_2 .

Now, the following results can be shown:

- If the graph resulting from the above construction is cyclic and a cycle includes p and $\neg p$ for some literal p , the formula is unsatisfiable.
- If this situation does not arise, we can perform a value assignment to the variables (find a satisfying assignment) as follows:
 - If a literal x (for variable x) is followed by literal $\neg x$ in the graph, assign x false.
 - If a literal $\neg x$ (for variable x) is followed by x in the graph, assign x true.
 - Else assign x arbitrarily.

Chapter Summary

This chapter considered many topics in propositional and first order logic. The highlights were: (i) the undecidability of the validity of first order logic sentences, by reduction from PCP, and (ii) Boolean satisfiability from a theoretical and practical point of view. Armed with this background, we will next study the very important problem of NP-completeness.

Exercises

18.1. Using the operators used in the Hilbert style axiomatization of Section 18.1, describe how to implement the familiar operators \wedge , \vee , and \oplus (exclusive-OR). Also implement the constants *true* and *false*.

18.2.

1. Describe what the following terms mean:
 - a) axiom
 - b) rule of inference
 - c) theorem
 - d) satisfiable
 - e) proof
 - f) tautology
2. Show that the following is a tautology:

$$(\neg x \Rightarrow \neg y) \Leftrightarrow (y \Rightarrow x)$$

Recall that $x \Leftrightarrow y$ means $(x \Rightarrow y) \wedge (y \Rightarrow x)$.

18.3. Prove the following using a similar approach as illustrated above (Sperschneider's book, p. 107): *Every barber shaves everyone who does not shave himself. No barber shaves someone who shaves himself. Prove that there exists no barber.*

18.4. Show that *Fmla2* is true under Interpretation 2 of page 328, describing the witness function (" $\exists F$ "). Also show that it is false under Interpretation 3.

18.5. One often falls into the following trap of claiming that FOL formulas are decidable: the negation of every FOL formula is an FOL formula; thus one can enumerate proofs, and see whether F or $\neg F$ gets enumerated in a proof, and hence decide the validity of F . What is the fallacy in this argument? (Hint: Are the cases listed, namely F being valid and $\neg F$ being unsatisfiable, exhaustive)?

18.6. Argue that the set of all non-valid but satisfiable FOL formulas is not recursively enumerable.

18.7.

1. Minimize the Boolean expression $(\neg a + ab)c + ac$ using a Karnaugh map.
2. How many Boolean functions can you build with a 4-to-1 multiplexer that has two select inputs?

18.8. Modify the program in Figures 18.2 and 18.3 to obtain a CNF to DNF converter. Test your resulting program.

18.9. Argue that DNF satisfiability and CNF validity have linear-time algorithms.

18.10. Solve Exercise 5.29 through proof by contradiction. Encode the problem as a CNF formula, and use a SAT tool.

18.11. Solve Exercise 5.30 similar to Exercise 18.10.

18.12. Write down the CNF formula represented by the DIMACS file given in Figure 18.4.

18.13. Prove that this transformation is indeed a polynomial-time mapping reduction \leq_P from 3-SAT to \neq -sat. In other words, prove that (i) it is a mapping reduction, and (ii) the runtime of the function involved is a polynomial with respect to the input size.

18.14. Why does the conversion from general CNF to 3-CNF described on page 341 preserve only satisfiability? (i.e., why do the formulas not emerge to be logically equivalent)

18.15. Prove that the 2-CNF satisfiability algorithm sketched in Section 18.3.8 is correct (meaning that it finds a satisfying assignment whenever one exists).

18.16. Apply the 2-CNF satisfiability algorithm of Section 18.3.8 to the following 2-CNF formulas, showing the steps as well as your final conclusion as to the satisfiability of these formulas:

- $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b)$
- $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$

