16 Provability predicates and the unprovability of consistency

We learned in the last chapter that no consistent extension of Q was decidable, and that any complete axiomatizable theory was decidable; we concluded that no consistent, axiomatizable extension of Q was complete. In the present chapter we are going to discuss a certain theory, *Elementary Peano Arithmetic*, or Z, as we shall call it (following Hilbert-Bernays). Z's language is L, the language of arithmetic. The *'non-logical' axioms of Z are* $Q1$ through $Q7$, the seven axioms of Q, together with all *induction axioms*: sentences obtained from formulas of L of the form:

$$([A(\mathbf{o}) \,\&\, \forall x\,(A(x) \to A(x'))] \to \forall x\, A(x))$$

by the prefixing of universal quantifiers. The theorems of Z are the logical consequences in L of the axioms of Q and the induction axioms. ($Q3$ actually follows from an induction axiom. For let

$$A(x) = (x \neq \mathbf{o} \to \exists y\, x = y').$$

Then $A(\mathbf{o})$ and $\forall x\,(A(x) \to A(x'))$ are logical truths; and so

$$\forall x\, A(x), = Q3, = \forall x\,(x \neq \mathbf{o} \to \exists y\, x = y'),$$

is a theorem of Z, and thus its inclusion as a non-logical axiom of Z was not necessary.) Like the axioms of Q, the induction axioms are all true in the standard interpretation $\mathcal{N}$, and hence Z is consistent; Z is even more evidently an axiomatizable extension of Q. Z is thus incomplete.

But though incomplete, because of the presence of the induction axioms, Z is a much more powerful theory than Q: all of the sentences mentioned in Exercise 14.2, for example, are theorems of Z but not of Q. And if Z is supplied with a notion of *proof* (such as the one given below), then the proofs of a great many mathematical theorems, including much (notably) of the theory of numbers, can be 'reproduced' or 'carried out' in Z. Unlike Q, Z is a theory in which large portions of actual mathematics can be adequately formalized. And, as Z is an extension of Q, all recursive functions (sets or relations) are representable (definable) in Z.

In the present chapter we are going to investigate two closely related questions about Z and other theories: whether the consistency of Z is provable in Z, and whether a certain sentence, which may be taken to

assert its own provability (in Z) is in fact true (and thus provable) or false (and thus unprovable). Unlike the question whether a sentence expressing its own unprovability (in Z) is provable or not – any such sentence must be both true and unprovable if everything provable is true – there seems to be no easy way to decide this second question *a priori*.

We shall now introduce, informally, the notion of a *proof of* a sentence in Z. We shall assume, but not prove, this fact: there is an effectively specifiable set of valid sentences, the 'logical' axioms of Z, such that a sentence A is a theorem of Z if and only if there is a finite sequence of sentences (of L) ending with A, each of which is either an axiom of Z (logical or non-) or the 'ponential' of two *earlier* sentences in the sequence. (D is the 'ponential' of C and $(C \to D)$.) Such a sequence shall be called a *proof in Z of A*.

We shall be a little bit more specific about the 'nature' of finite sequences of sentences: each such sequence is to be identified with the expression consisting of the same sentences in the same order, but separated by commas. As the comma has already been assigned the gödel number 29, every proof in Z acquires its own gödel number in consequence of this identification. The relation Proof, $= \{\langle m, n\rangle \mid m$ is the gödel number of a proof in Z of the sentence with gödel number $n\}$, is thus a recursive relation and is therefore definable in Z.

By 'straightforwardly transcribing' in L the definition of *proof in Z of* just given, making reference to gödel numbers instead of expressions, and utilizing, where necessary, the β-function of Chapter 14, we can construct a formula $\mathrm{Pr}(x,y)$ of L, which not only defines Proof in Z, but possesses certain other important properties as well. In order to describe these, we need a

Definition

$\mathrm{Prov}(y)$ is to be the formula $\exists x\, \mathrm{Pr}(x,y)$.

One important property that $\mathrm{Pr}(x,y)$ and $\mathrm{Prov}(y)$ have is that for any sentences A and C (of L),

(i) if $\vdash_Z A$, then $\vdash_Z \mathrm{Prov}(\ulcorner A \urcorner)$;
(ii) $\vdash_Z \mathrm{Prov}(\ulcorner A \to C \urcorner) \to [\mathrm{Prov}(\ulcorner A \urcorner) \to \mathrm{Prov}(\ulcorner C \urcorner)]$; and
(iii) $\vdash_Z \mathrm{Prov}(\ulcorner A \urcorner) \to \mathrm{Prov}(\ulcorner \mathrm{Prov}(\ulcorner A \urcorner) \urcorner)$.

(Later we shall express this property of $\mathrm{Prov}(y)$ by saying that $\mathrm{Prov}(y)$ is a *provability predicate* for Z. The consequent of the sentence mentioned in (iii) is the result of substituting $\mathbf{k}$ for y in $\mathrm{Prov}(y)$, k being the gödel images, whereas arithmetic operations apply to multivalued pixels. Logic operations are basic tools in binary image processing, where they are used for tasks such as masking, feature detection, and shape analysis. Logic operations on entire images are performed pixel by pixel. Because the AND operation of two binary variables is 1 only when both variables are 1, the result at any location in a resulting AND image is 1 only if the corresponding pixels in the two input images are 1. As logic operations involve only one pixel location at a time, they can be done in place, as in the case of arithmetic operations. Figure 2.14 shows various examples of logic operations, where black indicates 1 and white indicates 0. The XOR (exclusive OR) operation yields a 1 when one or the other pixel (but *not* both) is 1, and it yields a 0 otherwise. This operation is unlike the OR operation, which is 1 when one or the other pixel is 1, or both pixels are 1.

In addition to pixel-by-pixel processing on entire images, arithmetic and logic operations are used in neighborhood-oriented operations. Neighborhood processing typically is formulated in the context of so-called *mask* operations (the terms *template*, *window*, and *filter* also are often used to denote a mask). The idea behind mask operations is to let the value assigned to a pixel be a function of its gray level and the gray level of its neighbors. For instance, consider the subimage area shown in Fig. 2.15(a), and suppose that we want to replace the value of z_5 with the average value of the pixels in a 3 × 3 region centered at the pixel with value z_5. To do so entails performing an arithmetic operation of the form

$$z = \frac{1}{9}(z_1 + z_2 + \cdots + z_9) = \frac{1}{9}\sum_{i=1}^{9} z_i$$

and assigning to z_5 the value of z.

With reference to the mask shown in Fig. 2.15(b), the same operation can be obtained in more general terms by centering the mask at z_5 multiplying each pixel under the mask by the corresponding coefficient, and adding the results; that is,

$$z = w_1 z_1 + w_2 z_2 + \cdots + w_9 z_9 = \sum_{i=1}^{9} w_i z_i \tag{2.4-5}$$

If we let $w_i = 1/9$, $i = 1, 2, \ldots, 9$, this operation yields the same result as the averaging procedure just discussed.

Equation (2.4-5) is used widely in image processing. Proper selection of the coefficients and application of the mask at each pixel position in an image makes possible a variety of useful image operations, such as noise reduction, region thinning, and edge detection. However, applying a mask at each pixel location in an image is a computationally expensive task. For example, applying a 3 × 3 mask to a 512 × 512 image requires nine multiplications and eight

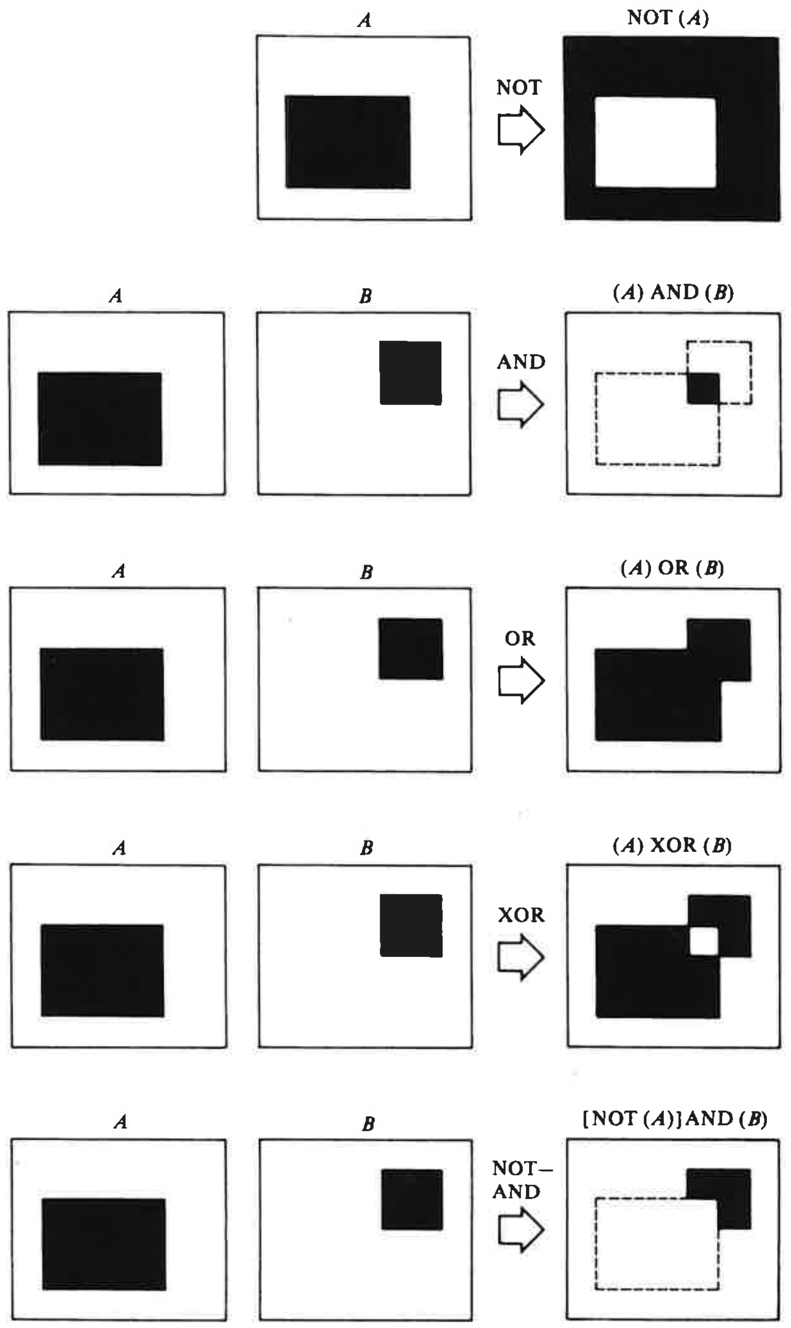
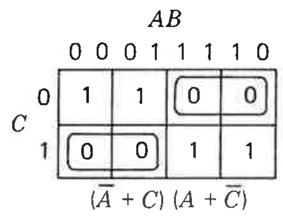
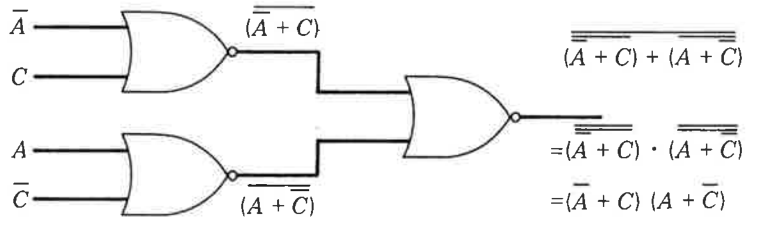


Figure 2.14 *Some examples of logic operations on binary images.*

A	B	C	OUTPUT	SUM TERMS
0	0	0	1	$A+B+C$
0	0	1	0	$A+B+\bar{C}$
0	1	0	1	$A+\bar{B}+C$
0	1	1	0	$A+\bar{B}+\bar{C}$
1	0	0	0	$\bar{A}+B+C$
1	0	1	1	$\bar{A}+B+\bar{C}$
1	1	0	0	$\bar{A}+\bar{B}+C$
1	1	1	1	$\bar{A}+\bar{B}+\bar{C}$

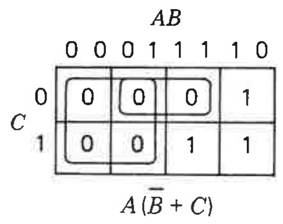


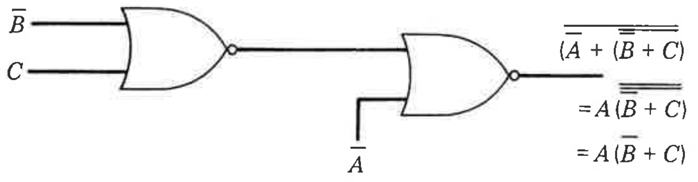
$(A+B+C)\,(A+\bar{B}+C)\,(\bar{A}+B+\bar{C})\,(\bar{A}+\bar{B}+\bar{C})$



$\bar{A}$
C
$\overline{(\bar{A}+C)}$
A
$\bar{C}$
$\overline{(A+\bar{C})}$
$\overline{\overline{(\bar{A}+C)}+\overline{(A+\bar{C})}}$
$=\overline{\overline{(\bar{A}+C)}}\cdot\overline{\overline{(A+\bar{C})}}$
$=(\bar{A}+C)\,(A+\bar{C})$

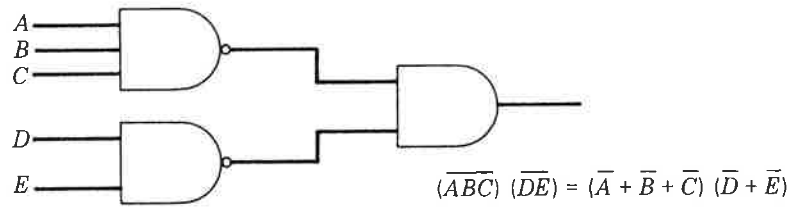
A	B	C	OUTPUT	SUM TERMS
0	0	0	0	$A+B+C$
0	0	1	0	$A+B+\bar{C}$
0	1	0	0	$A+\bar{B}+C$
0	1	1	0	$A+\bar{B}+\bar{C}$
1	0	0	1	$\bar{A}+B+C$
1	0	1	1	$\bar{A}+B+\bar{C}$
1	1	0	0	$\bar{A}+\bar{B}+C$
1	1	1	1	$\bar{A}+\bar{B}+\bar{C}$





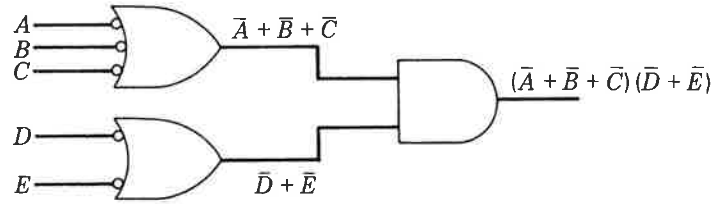
$\bar{B}$
C
$\bar{A}$
$\overline{(\bar{A}+\overline{(\bar{B}+C)})}$
$=A\,\overline{\overline{(\bar{B}+C)}}$
$=A(\bar{B}+C)$

FIG. 3 • 32 Two NOR gate designs.



A
B
C
D
E
$(\overline{ABC})\,(\overline{DE})=(\bar{A}+\bar{B}+\bar{C})\,(\bar{D}+\bar{E})$

a. Conventional NAND to AND gate network



A
B
C
$\bar{A}+\bar{B}+\bar{C}$
$(\bar{A}+\bar{B}+\bar{C})(\bar{D}+\bar{E})$
D
E
$\bar{D}+\bar{E}$

b. NAND-to-AND in (a) but with equivalent gates substituted for NANDs.

FIG. 3 • 33 NAND-to-AND gate networks. (*a*) Conventional NAND-to-AND gate network. (*b*) NAND-to-AND with equivalent gates substituted.

Since NAND gates are quite popular, and since the outputs from NAND gates can sometimes be ANDed by a simple connection, as will be shown, it is desirable to have analysis and design procedures for NAND-to-AND gate networks. To facilitate this, we again use our equivalent NAND gate symbol from Fig. 3·24, redrawing Fig. 3·33a as shown in Fig. 3·33b. This figure mirrors the Boolean algebra rule

$$(\overline{A\cdot B\cdot C})\cdot(\overline{D\cdot E}) = (\bar{A}+\bar{B}+\bar{C})\cdot(\bar{D}+\bar{E})$$

Examination of Fig. 3·33b shows that a NAND-to-AND gate network performs the same function as an OR-to-AND gate network, but with each input complemented. This gives us a design rule.

Design rule

To design a NAND-to-AND gate network use the procedure for an OR-to-AND gate network, then draw the block diagram using a NAND-to-AND form, but compliment each input.

Example: Design a NAND-to-AND gate network for the conditions given in Table 3·24.

We add the sum term column (Table 3·25).

†NAND gates with this property are designated as such by the integrated-circuit manufacturer. Not all TTL NAND

subprograms. In a later section of this chapter we shall discuss functions and procedures and the verification of larger programs involving such structures once they themselves have been verified. Firstly, however, we state an obvious metatheorem which we shall use frequently.

Theorem 5.1

For the implemented interpretation
(i) If $\vdash_M I \Rightarrow J$ and $\{J\}S\{O\}$ is partially verified, then $\{I\}S\{O\}$ is partially verified.
(ii) If $\{I\}S\{J\}$ is partially verified and $\vdash_M J \Rightarrow O$, then $\{I\}S\{O\}$ is partially verified.

In this chapter we shall use a widely adopted notation for proof rules which takes the following form:

$$\frac{\{I_1\}S_1\{O_1\}, \{I_2\}S_2\{O_2\}, \ldots, \{I_n\}S_n\{O_n\}, R_1, R_2, \ldots, R_m}{\{I\}S\{O\}}$$

This is shorthand for 'if $\{I_1\}S_1\{O_1\}$, $\{I_2\}S_2\{O_2\}$, ..., $\{I_n\}S_n\{O_n\}$ are partially verified and the formulae $R_1, R_2, \ldots, R_m$ are all provable in the interpreted implementation, then $\{I\}S\{O\}$ is partially verified'. Theorem 5.1 provides us with the following two proof rules expressed in this notation:

$$\frac{\{J\}S\{O\}, I \Rightarrow J}{\{I\}S\{O\}}$$

and

$$\frac{\{I\}S\{J\}, J \Rightarrow O}{\{I\}S\{O\}}$$

A similar proof rule, which is left for the reader to establish as an exercise, is the *combining rule*

$$\frac{\{I_1\}S\{O_1\}, \{I_2\}S\{O_2\}, I \Rightarrow I_1 \wedge I_2, O_1 \wedge O_2 \Rightarrow O}{\{I\}S\{O\}}$$

The proof rule associated with the assignment $x := t$ is the *assignment rule*

$$\frac{I \Rightarrow O[x|t]}{\{I\}x := t\{O\}}$$

Assuming a is an integer, we can use this assignment rule to deduce from $a>0 \Rightarrow a-1 \geqslant 0$ that $\{a>0\}$ $x:=a-1$ $\{x \geqslant 0\}$. Also, since $a-1=a-1$ trivially holds, we have true$\Rightarrow a-1=a-1$ and hence the assignment rule allows us to partially verify $\{\text{true}\}$ $x:=a-1$ $\{x=a-1\}$. Then we can use the combining rule to partially verify $\{a>0\}$ $x:=a-1$ $\{x \geqslant 0 \wedge x=a-1\}$. Now, we consider the various ways in which Pascal enables us to construct complex statements from simpler components.

(1) The serial statement

If $S_1, S_2, \ldots, S_n$ $(n \geqslant 1)$ are statements, then so is their serial composition

begin $S_1; S_2; \ldots; S_n$ **end**

To partially verify

$\{I\}$ **begin** $S_1; S_2; \ldots; S_n$ **end** $\{O\}$

we need formulae $J_1, J_2, \ldots, J_{n-1}$ and partial verifications of $\{I\}S_1\{J_1\}$, $\{J_1\}S_2\{J_2\}$, ..., $\{J_{n-1}\}S_n\{O\}$. The flowchart describing this is given in Fig. 5.2.

The proof rule for serial statements is

$$\frac{\{I\}S_1\{J_1\}, \{J_1\}S_2\{J_2\}, \ldots, \{J_{n-1}\}S_n\{O\}}{\{I\}\ \textbf{begin}\ S_1; S_2; \ldots; S_n\ \textbf{end}\ \{O\}}$$

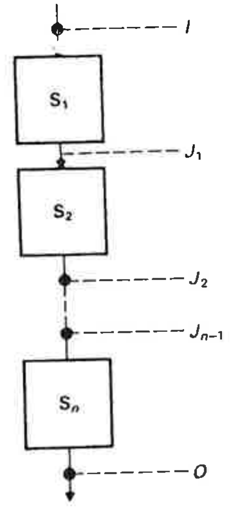


Fig. 5.2.