Computability, Complexity, and Languages

By Martin D. Davis et al

First Edition

CONTENTS

CHAPTER 1

PRELIMINARIES

1 Sets and n-tuples

We shall often be dealing with *sets* of objects of some definite kind. Thinking of a collection iof entities as a *set* simply amounts to a decision to regard the whole collection as a single object. We shall use the word *class* as synonymous with *set.* In particular we write N for the set of *natural numbers* $0, 1, 2, 3 \cdots$.

It is useful to speak of the *empty set*, written $\varnothing$, which has no members. The equation $R = S$, where R and S are sets, means that R and S are *identical as sets*, that is, that they have exactly the same members. We write $R \subseteq S$ and speak of R as a *subset* of S to mean that every element of R is also an element of S. We write $R \subset S$ to indicate that $R \subseteq S$ but $R \neq S$. In this case R is called a *proper subset* of S. If R and S are set, we write $R \cup S$ for the *union* of R and S, which is the collection of all objects which are members of either R or S or both. $R \cap S$, the *intersection* of R and S, is the set of all objects that belong to both R and S. $R - S$, the set of all objects that belong to R and do not belong to S, is the *difference* between R and S. Often we will be working in contexts where all sets being considered are subsets of some fixed set D (sometimes called a *domain* or a *universe*). In such a case we write $\bar{S}$ for $D - S$, and call $\bar{S}$ the *complement* of S. We write

$$\{a_1, a_2, \cdots, a_n\}$$

for the set consisting of the n objects $a_1, a_2, \cdots, a_n$. Sets that can be written in this form as well as the empty set are called *finite.* Sets that are not finite are called *infinite.* Since two sets are equal if and only if they have the same members. That is, the order in which we may choose to write the members of a set is irrelevant. Where order is important, we speak instead of an n-tuple or a *list.* A 2-tuple is called an *ordered pair*, and a 3-tuple is called an *ordered triple.* Unlike the case for sets of one object, we *do not distinguish between the object a and the 1-tuple* (a). The crucial property of n-tuples is

$$(a_1, a_2, \cdots, a_n) = (b_1, b_2, \cdots, b_n)$$

if and only if

$$a_1 = b_1, \quad a_2 = b_2, \quad \ldots, \quad \textit{and} \quad a_n = b_n.$$

If $S_1, S_2, \cdots, S_n$ are given sets, then we write $S_1 \times S_2 \times \cdots \times S_n$ for the set of all n-tuples such that $a_1 \in S_1, a_2 \in S_2, \cdots, a_n \in S_n$. $S_1 \times S_2 \times \cdots \times S_n$ is sometimes called the *Cartesian product* of $S_1, S_2, \cdots, S_n$.

2 Functions

For f a function, one writes $f(a) = b$ to mean that $(a, b) \in f$; the definition of function ensures that for each a there can be at most one such b. The set of all a such that $(a, b) \in f$ for some b is called the *domain* of f. The set of all $f(a)$ for a in the domain of f is called the *range* of f.

Functions f are often specified by *algorithms* that provide procedures for obtaining $f(a)$ from a. However, it is quite possible to possess an algorithm that specifies a function without being able to tell which elements belong to its domain. This makes the notion of a so-called *partial function* play a central role in computability theory. A *partial function on a set S* is simply a function whose domain is a subset of S. If f is a partial function on S and $a \in S$, then we write $f(a)\downarrow$ and say that $f(a)$ is *defined* to indicate that a is in the domain of f; if a is not in the domain of f, we write $f(a)\uparrow$ and say that $f(a)$ is *undefined*. If a partial function on S has the domain S, then it is called *total*. Finally, we should mention that the empty set $\varnothing$ is itself a function. Considered as a partial function on some set S, *it is nowhere defined*.

A partial function f on a set S^n is called an *n-ary partial function on* S, or a function of n variables on S. We use *unary* and *binary* for 1-ary and 2-ary, respectively.

A function f is *one-one* if, for all x, y in the doamin of f, $f(x) = f(y)$ implies $x = y$. If the range of f is the set S, then we say that f is an *onto* function with respect to S, or simply that f is *onto* S.

We will sometimes refer to the idea of *closure*. If S is a set and f is a partial function on S, then S is *closed under* f if the range of f is a subset of S.

3 Alphabets and Strings

An *alphabet* is simply some finite nonempty set A of objects called *symbols*. An n-tuple of symbols of A is called a *word* or a *string* on A. The set of all words on the alphabet A is written A^*. Any subset of A^* is called a *language on* A or a *language with alphabet* A. We do *not* distinguish between a symbol $a \in A$ and the word of length 1 consisting of that symbol.

4 Predicates

By a *predicate* or a *Boolean-valued function* on a set S we mean a *total* function P on S such that for each $a \in S$, either

$$P(a) = \text{TRUE} \quad \text{or} \quad P(a) = \text{FALSE},$$

where TRUE and FALSE are a pair of distinct objects called *truth values*. We often say $P(a)$ *is true* for $P(a)$ =TRUE, and $P(a)$ *is false* for $P(a)$ =FALSE. Given a predicate P on a set S, there is a corresponding subset R of S, namely, the set of all elements $a \in S$ for which $P(a) = 1$. The predicate P is called the *characteristic function* of the set R.

5 Quantifiers

In this section we will be concerned exclusively with predicates on N^m (or what is the same thing, m-ary predicates on N) for different values of m. Thus, let $P(t, x_1, \cdots, x_n)$ be an $(n+1)$-ary predicate. Consider the predicate $Q(y, x_1, \cdots, x_n)$ defined by

$$\begin{aligned} Q(y, x_1, \ldots, x_n) \Leftrightarrow & P(0, x_1, \ldots, x_n) \vee P(1, x_1, \ldots, x_n) \\ & \vee \cdots \vee P(y, x_1, \ldots, x_n). \end{aligned}$$

Thus the predicate $Q(y, x_1, \cdots, x_n)$ is true just in case there is value of $t \leq y$ such that $P(t, x_1, \cdots, x_n)$ is true. We write this predicate Q as

$$(\exists t)_{\leq y} P(t, x_1, \ldots, x_n).$$

The expression "$(\exists t)_{\leq y}$" is called a *bounded existential quantifier*. Similarly, we write $(\forall t)_{\leq y} P(t, x_1, \ldots, x_n)$ for the predicate

$$P(0, x_1, \ldots, x_n) \& P(1, x_1, \ldots, x_n) \& \cdots \& P(y, x_1, \ldots, x_n).$$

The predicate is true just in case $P(t, x_1. \cdots, x_n)$ is true for *all* $t \leq y$. The expression "$(\forall t)_{\leq y}$" is called a *bounded universal quantifier*.

6 Proof by Contradiction

Recall that a number is called a *prime* if it has *exactly two distinct divisors*, itself and 1. Consider the following assertion:

$$n^2 - n + 41 \text{ is prime for all } n \in N.$$

This assertion is in fact *false*.

In a *proof by contradiction*, one begins by supposing that the assertion we wish to prove is false. In a proof by contradiction we look for a pair of statements developed in the course of the proof which *contradict* one another.

Theorem 6.1

Let $x \in \{a, b\}^*$ such that $xa = ax$. Then $x = a^{[n]}$ for some $n \in N$.

7 Mathematical Induction

Mathematical induction furnishes an important technique for proving statements of the form $(\forall n)P(n)$, where P is a predicate on N. One proceeds by proving a pair of auxiliary statements, namely, $P(0)$ and

$$(\forall n)(\textit{if } P(n) \textit{ then } P(n+1)). \tag{1.1}$$

Why is this helpful? Because sometimes it is much easier to prove (1.1) than to prove $(\forall n)P(n)$ in some other way. In proving this second auxiliary proposition one typically considers some fixed but arbitrary value k of n and shows that if we assume $P(k)$ we can prove $P(k+1)$. $P(k)$ is then called the *induction hypothesis*.

There are some paradoxical things about proofs by mathematical induction. One is assuming $P(k)$ for some *particular* k in order to show that $P(k+1)$ follows.

It is also paradoxical that in using induction (we shall often omit the word *mathematical*), it is sometimes easier to prove statements by first making them "stronger." We wish to prove $(\forall n)P(n)$. Instead we decide to prove the *stronger* assertion $(\forall n)(P(n) \& Q(n))$ (which of course implies the original statement). The technique of deliberately strengthening what is to be proved for the purpose of making proofs by induction easier is called *induction loading*.

Theorem 7.1

For all $n \in N$ we have $\sum_{i=0}^{n}(2i+1) = (n+1)^2$.

Another form of mathematical induction that is often very useful is called *course-of-values induction* or sometimes *complete induction*.

Theorem 7.2

There is no string $x \in \{a, b\}^*$ such that $ax = xb$.

CHAPTER 2

PROGRAMS AND COMPUTABLE FUNCTIONS

1 A Programming Language

In particular, the letters

$$X_1\ X_2\ X_3\ \cdots$$

will be called the *input variables* of φ, the letter Y will be called the *output variable* of φ, and the letters

$$Z_1\ Z_2\ Z_3\ \cdots$$

will be called the *local variables* of φ.

In φ we will be able to write "instructions" of various sorts; a "program" of φ will then consist of a *list* (i.e., a finite sequence) of instructions.

Table 2.1

Insturction	Interpretation
$V \leftarrow V + 1$	Increase by 1 the value of the variable V.
$V \leftarrow V - 1$	If the value of V is 0, leave it unchanged; otherwise decrease by 1 the value of V.
IF $V \neq 0$ GOTO L	If the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the list

We give in Table 2.1 a complete list of our instructions. In this list V stands for any variable and L stands for any label.

These instructions will be called the *increment*, *decrement*, and *conditional branch* instructions, respectively.

We will use the special convention that *the output variable Y and the local variables Z_i initially have the value* 0.

2 Some Examples of Programs

Our first example is the program

$$\begin{array}{ll} [A] & X \leftarrow X - 1 \\ & Y \leftarrow Y + 1 \\ & \text{IF } X \neq 0 \text{ GOTO } A \end{array}$$

If the initial value x of X is not 0, the effect of this program is to copy x into Y and to decrement the value of X down to 0. We will say that this program *computes* the function

$$f(x) = \begin{cases} 1 & \text{if} \quad x = 0 \\ x & \text{otherwise.} \end{cases}$$

Although the preceding program is a perfectly well-defined program of our language φ, we may think of it as having arisen in an attempt to write a program that copies the value of X into Y, and therefore containing a "bug" because it does not handle 0 correctly. The following slightly more complicated example remedies this situation.

$$\begin{array}{ll}
[A] & \text{IF } X \neq 0 \text{ GOTO } B \\
 & Z \leftarrow Z + 1 \\
 & \text{IF } Z \neq 0 \text{ GOTO } E \\
[B] & X \leftarrow X - 1 \\
 & Y \leftarrow Y + 1 \\
 & Z \leftarrow Z + 1 \\
 & \text{IF } Z \neq 0 \text{ GOTO } A
\end{array}$$

At first glance Z's role in the computation may not be obvious. It is used simply to allow us to code an *unconditional branch*. That is, the program segment

$$\begin{array}{l}
Z \leftarrow Z + 1 \\
\text{IF } Z \neq 0 \text{ GOTO } L
\end{array} \tag{2.1}$$

has the effect (ignoring the effect on the value of Z) of an instruction

$$\text{GOTO } L$$

such as is available in most programming languages. Now GOTO L is not an instruction in our language φ, but since we will frequently have use for such an instruction, we can use it as an abbreviation for the program segment (2.1). Such an abbreviating pseudoinstruction will be called a *macro* and the program or program segment which it abbreviates will be called it *macro expansion*.

For our final example, we take the program

$$\begin{array}{ll}
 & Y \leftarrow X_1 \\
 & Z \leftarrow X_2 \\
[C] & \text{IF } Z \neq 0 \text{ GOTO } A \\
 & \text{GOTO } E \\
[A] & \text{IF } Y \neq 0 \text{ GOTO } B \\
 & \text{GOTO } A \\
[B] & Y \leftarrow Y - 1 \\
 & Z \leftarrow Z - 1 \\
 & \text{GOTO } C
\end{array}$$

What happens if we begin with a value of X_1 less than the value of X_2? At this point the computation enters the "loop":

$$\begin{array}{ll}
[A] & \text{IF } Y \neq 0 \text{ GOTO } B \\
 & \text{GOTO } A
\end{array}$$

Since $y = 0$, there is no way out of this loop and the computation will continue "forever." Thus, if we begin with $X_1 = m$, $X_2 = n$, where $m < n$, the computation will never terminate. In this case (and in similar cases) we will say that the program computes the *partial function*

$$g(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{if } \quad x_1 \geq x_2 \\ \uparrow & \text{if } \quad x_1 < x_2. \end{cases}$$