THE INTRINSIC COMPUTATIONAL DIFFICULTY OF FUNCTIONS

ALAN COBHAM
I.B.M. Research Center, Yorktown Heights, N. Y., U.S.A.

The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? I grant I have put the first of these questions rather loosely; nevertheless, I think the answer, ought to be: *yes*. It is the second, which asks for a justification of this answer which provides the challenge.

The difficulty does not stem from the fact that the first question has been imprecisely formulated. There seems to be no substantial problem in showing that using the standard algorithms it is in general harder—in the sense that it takes more time or more scratch paper—to multiply two decimal numbers than to add them. But this does not answer the question, which is concerned with the computational difficulty of these two operations without reference to specific computational methods; in other words, with properties intrinsic to the functions themselves and not with properties of particular related algorithms. Thus to complete the argument, I would have to show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block. Of course, as I have implied, I feel this must be the case and that multiplication is in this absolute sense harder than addition, but I am forced to admit that at present my reasons for feeling so are of an essentially extra-mathematical character.

Questions of this sort belong to a field which I think we might well call *metanumerical-analysis*. The name seems appropriate not only because it is suggestive of the subject matter, namely, the methodology of computation, but also because the relationship of the field to computation is closely analogous to that of metamathematics to mathematics. In metamathematics, we encounter problems concerned with specific proof systems; e.g., the existence of proofs having a certain form, or the adequacy of a system in a given context. We also encounter problems concerned with provability but independently of any particular proof system; e.g., the undecidability of mathematical theories. So in metanumerical-analysis we encounter problems related to specific computational systems or categories of computing machines as well as problems such as those mentioned above which, though concerned with computation, are independent of any particular method of computation. It is this latter segment of metanumerical-analysis I would like to look at more closely.

Let me begin by stating two of the very few results which fall squarely within this area. (Others appear in references [3], [4] and [5].) Both are drawn from Ritchie's work on predictably computable functions [6], the first being an almost immediate generalization of one part of that work, the second an incomplete rendering of another part. These relate, perhaps not in the most happy fashion, the computational complexity of a function with its location in the Grzegorczyk hierarchy [2]. Recall that this hierarchy is composed of a sequence of classes:

$$\mathscr{E}^0 \subset \mathscr{E}^1 \subset \mathscr{E}^2 \subset \dots,$$

each properly contained in the next, and having as union the class of all primitive recursive functions. $\mathscr{E}^2$ can be characterized as the smallest class containing the successor and multiplication functions, and closed under the operations of explicit transformation, composition and limited recursion. The classes $\mathscr{E}^3$ (which is the familiar class of Kalmar elementary functions) and $\mathscr{E}^4$ have similar characterizations, with exponentiation and the function $\underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{y}$ respectively, replacing multiplication in the characterization of $\mathscr{E}^2$. The higher classes can be characterized inductively in like manner.

With each single-tape Turing machine Z which computes a function of one variable we may associate two functions, σ_Z and τ_Z. Assuming some standard encoding of natural numbers—we might take decimal notation to be specific—we define $\sigma_Z(n)$, where n is a natural number, to be the numbers of steps (instruction executions) in the computation on Z starting with n encoded on its tape, and define $\tau_Z(n)$ to be the number of distinct tape squares scanned during the course of this computation. Restricting attention to functions of one variable, we have the following.

THEOREM. *For each $k \geqq 3$ the following five statements are equivalent:*

1. $f \ \varepsilon \mathscr{E}^k$;
2. *there exists a Turing machine Z which computes f and such that* $\sigma_Z \ \varepsilon \ \mathscr{E}^k$;
3. *there exists a Turing machine Z which computes f and a function g* $\varepsilon \ \mathscr{E}^k$ *such that, for all n,* $\sigma_Z(n) \leqq g(n)$;

4. – 5. *Same as* 2 – 3 *with* τ_Z *in place of* σ_Z.

This theorem has an immediate generalization to functions of several variables. From it we can infer that (in effect, but not quite precisely) if one function is simpler to compute than another, in the very strong sense that for any value of the argument the computation can be done in fewer steps or with less tape, then that function lies no higher than the other in the Grzegorczyk hierarchy. We cannot conclude the converse however: that if one function lies lower than another, it is necessarily simpler to compute. As a matter of fact, it appears that a function in the lower part of the hierarchy may actually be *on average* harder to compute than one higher up, though, of course, it cannot be harder for all values of the argument.

A word is needed as to why I have included a theorem involving Turing machines in a discussion which I said was going to be about method-independent aspects of computation. The fact is the theorem remains correct even if one considers far wider classes of computing machines. In particular, it holds for Turing machines with more than one tape or with multi-dimensional tapes providing the cells of the latter are arranged in reasonably orderly fashion. It also holds if the set of possible instructions is extended to include, e.g., erasure of an entire tape or resetting of a scanning head to its initial position (although I doubt such operations should be considered *steps* since it does not appear that they can be executed in a bounded amount of time).

The reason for such general applicability can be found on examination of the proof of this theorem. There we find that the fact that we are dealing with a particular class of Turing machines is quite incidental: it is the form of their arithmetization which counts. The geometry and basic operations of a Turing machine are of a sort which admit an arithmetization in which the functions which describe the step-by-step course of a computation on it are of a very simple nature, lying, in particular, well within the class $\mathcal{E}^2$. This is all that is needed to obtain the preceding theorem (as well as the one which follows). Now the class $\mathcal{E}^2$ is so rich in functions that it is almost inconceivable to me that there could exist *real* computers not having mathematical models whose arithmetization could be carried out in such a way that these associated functions would fall within it. Thus I suspect this theorem does indeed say something about the absolute computational properties of functions, and so fits properly in the discussion.

The five equivalences of the preceding theorem do not hold for $k < 3$. Ritchie has obtained a hierarchy which decomposes the range between $\mathcal{E}^2$ and $\mathcal{E}^3$ into classes of functions of varying degrees of computational difficulty; however, rather than go into this, I would like now to turn to the problem of classifying the functions within $\mathcal{E}^2$, where many of the functions most frequently encountered in computational work, addition and multiplication in particular, are located. First, concerning $\mathcal{E}^2$ itself, we have [6] the following.

THEOREM. *A function f belongs to $\mathcal{E}^2$ if and only if there exists a Turing machine Z which computes f and constants c_1 and c_2 such that $\tau_Z(n) \leqq c_1 l(n) + c_2$, for all n.*

Here $l(n)$ is the length of n, that is, the number of digits in its decimal representation. Machines which compute in the fashion described are equivalent to those which Myhill has called linear bounded automata [4]. Since merely writing n requires $l(n)$ tape squares, we must have $c_1 \geqq 1$. As a matter of fact, if we consider machines with arbitrarily large alphabets, then c_1 need only be enough larger than this to permit writing of the answer on the tape; e.g., if $f(n) = n^2$ we can take $c_1 = 2$; if $f(n) \leqq n$ for all n we can take $c_1 = 1$. In other words, if we have enough space to write the larger of the value and the argument of a function in $\mathcal{E}^2$ then we have enough space to carry out the entire computation. Consequently, the function τ is not a suitable tool for making fine distinctions concerning the computational difficulty of functions within $\mathcal{E}^2$. We might attempt to redefine what we mean by the amount of tape used during a computation by distinguishing between those locations used for writing input and output and those used in the actual computation. But the artificiality of such a seemingly ad hoc distinction would seem to be trending away from our goal of obtaining a natural analysis independent of the method or type of machine used in the computation.

This may be a good point to mention that, although I have so far been tacitly equating computational difficulty with time and storage requirements, I don't mean to commit myself to either of these measures. It may turn out that some measure related to the physical notion of work will lead to the most satisfactory analysis; or we may ultimately find that no single measure adequately reflects our intuitive concept of difficulty. In any case, for the present, I see no harm in restricting the discussion somewhat and, having discarded τ as a tool for reasons just stated, confining further attention to the analysis of computation time.

This leaves us some latitude for differentiating among functions in $\mathcal{E}^2$. Th closest analog of the foregoing theorem concerning σ, rather than τ, tha I know of states that for any f in $\mathcal{E}^2$ there exists a Turing machine Z whicl computes it and such that σ_Z is bounded by a polynomial in its argument f itself must also be bounded by a polynomial in its argument, but I don' know whether these two conditions in turn imply that f is in $\mathcal{E}^2$.

To obtain some idea as to how we might go about the further classifica tion of relatively simple functions, we might take a look at how we ordina rily set about computing some of the more common of them. Suppose, fo example, that m and n are two numbers given in decimal notation with on written above the other and their right ends aligned. Then to add m and n w start at the right and proceed digit-by-digit to the left writing down the sum. No matter how large m and n, this process terminates with the answer after a number of steps equal at most to one greater than the larger of $l(m)$ and $l(n)$. Thus the process of adding m and n can be carried out in a number of steps which is bounded by a linear polynomial in $l(m)$ and $l(n)$. Similarly, we can multiply m and n in a number of steps bounded by a quadratic polynomial in $l(m)$ and $l(n)$. So, too, the number of steps involved in the extraction of square roots, calculation of quotients, etc., can be bounded by polynomials in the lengths of the numbers involved, and this seems to be a property of simple functions in general. This suggests that we consider the class, which I will call $\mathscr{L}$, of all functions having this property.

For several reasons the class $\mathscr{L}$ seems a natural one to consider. For one thing, if we formalize the above definition relative to various general classes of computing machines we seem always to end up with the same well-defined class of functions. Thus we can give a mathematical characterization of $\mathscr{L}$ having some confidence that it characterizes correctly our informally defined class. This class then turns out to have several natural closure properties, being closed in particular under explicit transformation, composition and limited recursion on notation (digit-by-digit recursion). To be more explicit concerning the latter operation, which incidentally seems quite appropriate to computational work, we say that a function f is defined from functions $g, h_0, \ldots, h_9$, and k by limited recursion on notation (assuming decimal notation) if

$$f(x, 0) = g(x)$$
$$f(x, s_i(y)) = h_i(x, y, f(x, y)) \qquad (i = 0, \ldots, 9; \qquad i \neq 0 \text{ if } y = 0)$$
$$f(x, y) \leqq k(x, y),$$

where s_i is the generalized successor: $s_i(y) = 10y + i$. $\mathscr{L}$ is in fact the smallest class closed under these operations and containing the functions s_i and $x^{l(y)}$. It is closely related to, perhaps identical with, the class of what Bennett has called the extended rudimentary functions [1]. Since $\mathscr{L}$ contains $x^{l(y)}$, which cannot, by the second of the theorems mentioned earlier, belong to $\mathscr{E}^2$, $\mathscr{L}$ is not a subclass of $\mathscr{E}^2$. On the other hand, I strongly suspect that the function $f(n)$ = the nth prime, which is known to be in $\mathscr{E}^2$, does not belong to $\mathscr{L}$. If this is the case then $\mathscr{E}^2$ and $\mathscr{L}$ are incomparable and we have the unsurprising result that the categorization of the simpler functions as to computational difficulty yields divergent classifications according to the criterion of difficulty selected—in this case time and storage requirements. Concerning functions which are relatively simple under both criteria, that is, those in both $\mathscr{E}^2$ and $\mathscr{L}$, I can only offer further conjecture, namely that $\mathscr{E}^2 \cap \mathscr{L}$ ir a subclass of the constructive arithmetic functions, probably even

a proper subclass. (The function $f(n) = 1$ or 0, according as n is or is not prime, is constructive arithmetic but seemingly not in $\mathscr{L}$.)

An attempt to construct a natural computational hierarchy within $\mathscr{L}$ now brings out quite sharply one of the basic problems entailed in the study of absolute or intrinsic computational properties of functions. Suppose we start out in the obvious way and define, for each k, a subclass $\mathscr{L}^k$ of $\mathscr{L}$ consisting of all functions which can be computed in such a way that the number of steps in the computation is bounded by a polynomial of degree k in the lengths of the arguments. So defined, the classes $\mathscr{L}^k$ form an increasing sequence whose union is $\mathscr{L}$. Clearly, almost as a matter of definition, the analog of the theorem concerning the Grzegorczyk hierarchy I mentioned earlier will hold for this hierarchy: a function in the upper part of the hierarchy cannot be simpler to compute for every argument than one further down.

If we are to make any application of this theorem, we need a precise, mathematical characterization of the classes $\mathscr{L}^k$. Unlike the foregoing situation, however, we find that it makes a definite difference what class of computational methods and devices we consider in our attempt to formalize the definition. Thus, if we restrict attention to single-tape Turing machines, we find that addition does not belong to $\mathscr{L}^1$, whereas it does if we permit our machines to have several tapes. Similarly, multiplication gets into $\mathscr{L}^2$ only if we permit multi-tape machines. This certainly does not mean that there is no reasonable formalization of the classes of this hierarchy, but it does suggest that there may be some difficulty both in finding this formalization and, once found, in convincing oneself that it correctly captures all relevant aspects of the intuitive model.

The problem is reminiscent of, and obviously closely related to, that of the formalization of the notion of effectiveness. But the emphasis is different in that the physical aspects of the computation process are here of predominant concern. The question of what may legitimately be considered to constitute a step of a computation is quite unlike that of what constitutes an effective operation. I did not dwell particularly on what I consider to be the properties of legitimate step when I was discussing the classification of functions outside of $\mathscr{E}^2$ because, as I pointed out, one could admit all sorts of questionable operations as steps and, so long as they could be represented by functions in $\mathscr{E}^2$, the results obtained would remain unaltered. Quite similar remarks can be made concerning permissible geometric arrangements of the working area of a computation, and even concerning the types of notation used for representing natural numbers. If, however, we are to make fine distinctions, say between functions in $\mathscr{L}^1$ and functions in $\mathscr{L}^2$, then we must have an equally fine analysis of all phases of the computational process. It is no longer a problem of finding convincing arguments that every conceivable computing method can be arithmetized within $\mathscr{E}^2$ but rather of finding convincing arguments that these can somehow be arithmetized within whatever presumably more restricted class we settle upon as a formalization for $\mathscr{L}^1$. Of course, at the same time, we must be prepared to argue that we haven't taken too broad a class for $\mathscr{L}^1$, and thus admitted to it functions not in actuality computable in a number of steps linearly bounded by the lengths of its arguments. I think this is one of the fundamental problems of metanumerical-analysis and one whose resolution may well call for considerable patience and discrimination, but until it, and several related problems, have received more intensive treatment, I doubt we can find any really satisfying proof that multiplication is indeed harder than addition.

REFERENCES

[1] J. H. BENNETT, On Spectra, doctoral dissertation, Princeton University, 1962.

[2] A. GRZEGORCZYK, Some Classes of Recursive Functions, *Rozprawy Matematyczne*, 1953.

[3] J. HARTMANIS and R. E. STEARNS, On the Computational Complexity of Algorithms, Notes for the University of Michigan Summer Conference on Automata Theory, 1963, 59–79.

[4] J. MYHILL, Linear Bounded Automata, WADD Tech. Notes 60–165, University of Pennsylvania, Report No. 60–22, 1960.

[5] M. O. RABIN, Degree of Difficulty of Computing a Function and a Partial Ordering of Recursive Sets, Applied Logic Branch, Hebrew University, Jerusalem, Technical Report No. 2, 1960.

[6] R. W. RITCHIE, Classes of Predictably Computable Functions, *Trans. Amer. Math. Soc.* **106**, 1963, 139–173.

CLASSICAL EXTENSIONS OF INTUITIONISTIC MATHEMATICS[1]

S. C. KLEENE

University of Wisconsin, Madison, Wisconsin, U.S.A.

§ **1. Introduction.** Between the oral presentation of this paper and its publication, the speaker's and R. E. Vesley's monograph *The Foundations of Intuitionistic Mathematics* [10] (cited as "FIM") will have appeared. The original formalization of intuitionistic mathematics was by Heyting in 1930 [1, 2], using some specifically intuitionistic symbolism. In contrast, the symbolism in FIM is the same as that of a system of classical mathematics. This is accomplished by the use in FIM of type-1 function variables $\alpha, \beta, \gamma, \cdots$ (i.e. variables for one-place number-theoretic functions) for Brouwer's choice sequences, as well as for other uses of functions (i.e. particular functions or laws for Brouwer, and the functions of classical mathematics). There are also type-0 variables $a, b, c, \cdots, x, y, z, \cdots$ (ranging over the natural numbers $0, 1, 2, \cdots$), the logical symbols of a two-sorted predicate calculus, the equality symbol $=$, symbols for some particular primitive recursive functions and functionals (including 0), and Church's λ-operator (with number variables).[2]

The use of a common symbolism facilitates the study of relationships between intuitionistic and classical systems. Indeed, in FIM postulates are given for three formal systems: a *basic system B*, and divergent classical and intuitionistic systems. The *classical system C* arises from the basic system *B* by adding the law of double negation $\neg\neg A \supset A$ (or the law of the excluded middle $A \vee \neg A$) as an additional axiom schema, or in place of the intuitionistic negation-elimination postulate $\neg A \supset (A \supset B)$. The *intuitionistic system I* arises from the basic system by adding a postulate which expresses Brouwer's principle that, if to each (one-place number-theoretic) function α a number b is correlated, the correlated number b must be determined (under the operation of an algorithm) by some initial segment $\alpha(0), \cdots, \alpha(y-1)$ of the values of α. (Actually, we have postulated the generalization of this to the case that to each α a function β is correlated; cf. FIM § 7.)

---

[1] Part of the work reported herein was done in 1963-64 under Grant GP1624 from the National Science Foundation of the U.S.A.

[2] Many details must be left out in the half-hour talk. The reader of the published version will be able to consult FIM. The formalization in FIM was forecasted in [4, 6].