PLEASE READ | BY G. KREISEL

II. Checking of Computer Programs: An Example of Non-numerical Computation

We shall consider principally, but not exclusively, programs for solving some arithmetic problem with a parameter, n say. In terms of I §2(c) and I §3(a), the formal counterpart to the informal distinction between checking and proving (the validity of) a program is this:

Checking a program is a decidable procedure, by use of strongly computable rules; a check provides indeed a proof, by very elementary rules, of

(*) the function defined by the program solves the problem for all n.

In contrast, for the usual formal rules of inference, it is not (recursively) decidable whether or not a universal statement of the form (*) is derivable. This corresponds to the meaning of 'checking' as a procedure which does not require ingenuity (in contrast to 'cross checking' which does, but which need not constitute a complete check).

It will be best to begin with a closer look at the familiar ideas of 'arithmetic problem' and 'solution', including the choice of language used to present the solution; cf. I § 2(a). The reader should not forget for one moment that, in the context of checking, the program and our knowledge of its properties are the principal objects of study; we are not merely concerned with the sequence of states of the machine which the program, objectively, determines; and a fortiori, not merely with the graph of the function which the machine puts out; cf. footnote 3 of the Introduction.

1. Numerical analysis and programmation. The general form of an arithmetic problem (which we consider here) is

$$\exists m A(n,m) \quad \text{with parameter } n \,,$$

where the relation A may be defined by use of quite abstract concepts; for example, it may say that m is (a code for) an approximation, to accuracy $1/n$, to the solution, at the origin say, of some differential equation in Hilbert space.

Numerical analysis, by use of proofs not merely by mechanical checking, provides more or less explicit solutions in terms of a function f (of n)

determined in familiar terms, for example, 2^n or $[\log n]$ (the integral part of log n). One sometimes says that numerical analysis provides an algorithm; but, strictly speaking, the algorithm is only tacitly understood because in numerical estimates not the procedure of calculation but only the values of the function are relevant. In fact, numerical analysis provides in general not an algorithm but rather a simple functional equation for f which suggests an algorithm, e.g., in the case of 2^n

$$\forall p[f(0) = 1 \wedge f(p + 1) = 2f(p)] .$$

Numerical analysis has done its job when it finds such a functional property or equation $\Phi(f)$ and a proof of

$$\Phi(f) \rightarrow A(n,fn) .$$

(In §3c below we give some formal conditions ensuring that Φ does determine an algorithm.)

Programmation consists in making explicit, in mechanical terms, the algorithm 'suggested' by the property Φ. Roughly speaking, checking a proposed program π_f consists in establishing by the elementary methods implicit in the idea of checking that π_f satisfies Φ. Our problem here consists in making explicit what these elementary methods are. This involves of course some general conjecture on the type of properties Φ which occur in practice (but not on the type of relation A!).

The following example does not merely illustrate the difference between numerical analysis and programmation, but gives a good idea of the qualitative difference between a recursive and a feasible solution, discussed at length at the end of Note 1.

Instead of the single variable n, we consider the quadruples n_1, n_2, n_3 and n' and the relation

$$(m = 0 \;\&\; n_1^{2+n'} + n_2^{2+n'} = n_3^{2+n'}) \;\vee\; (m = 1 \;\&\; n_1^{2+n'} + n_2^{2+n'} \neq n_3^{2+n'}) .$$

Without numerical analysis, we have the obvious program of 'simply evaluating'

$$n_1^{2+n'} + n_2^{2+n'}, \text{ and } n_3^{2+n'}$$

for given positive integers n_1, n_2, n_3 and n' and 'comparing' them. The program must translate these instructions to the machine. With numerical analysis, that is a (partial or complete) proof of Fermat's conjecture, we should use a totally different program (for the same function of n_1, n_2, n_3, n'!), namely

$$n_1, n_2, n_3, n' \to 1 .$$

Amusingly, on the basis of current knowledge,[13] as far as existing computing machinery is concerned, only the second program is feasible (for numerical computing): for those numbers for which Fermat's conjecture is open at least one of the numbers $n_1^{2+n'}$, $n_2^{2+n'}$ and $n_3^{2+n'}$ is so large that it could not be stored in any machine's memory at all! A much more sophisticated example will be given in IV §2b.

2. Formalization of the concept of 'checking': definitional equality (brutal approach). To start with, I want to convey an idea for such a formalization which a programmer, relying on his judgment and experience, may occasionally decide to use. It is evident that any application will depend on the programming language, that is, what precisely the symbol π stands for in §1, and the possibly non-numerical terms (cf. I §2a) chosen to present the (numerical valued) solution in the particular application. I should try to refer the formalization to theories of programming languages or of (non-numerical) notation systems for natural numbers if I were convinced that any known theory is even remotely correct. As it is, the chance of introducing a systematic error by relying on current theories seems more damaging than the vagueness which results from giving only an illustration (as I do below). In the next section I shall try to discuss limitations.

Proposal. Assume that the critical properties Φ of §1 supplied by numerical analysis are conjunctions of equations $t_i = t_i'$ $(1 \leq i \leq N)$ between terms built up from f, the numerical variable p, and signs for specific programs (constants, successor operation, addition, multiplication, etc.). Assume further that, for any program π_f and any term $t[\pi]$ built up from π_f, p and the signs for specific programs (as above), a program is assigned to $t[\pi]$. Then we define:

π_f is checked for Φ if for each i, $1 \leq i \leq N$, the same programs are assigned to $t_i[\pi]$ and $t_i'[\pi]$.

The question whether or not the same program is assigned to two terms is supposed to be decidable, and this constitutes the elementary character of 'checking'.

Remark. It is familiar from recursion theory that for quite elementary formal systems, specifically any consistent r.e. extension of formal primitive arithmetic PRA, it is not decidable whether an equation between two terms containing a variable (p) is formally provable since the sets

$$\{t_1,t_2 : \underset{\text{PRA}}{\vdash} t_1 = t_2\} \quad \text{and} \quad \{t_1,t_2 : \exists n \underset{\text{PRA}}{\vdash} t_1[p/n] \neq t_2[p/n]\}$$

are recursively inseparable. (Here n ranges over numerical terms and $t[p/n]$ is the result of substituting n for the variable p in t.)

Illustrations. Probably the reader understands the proposal pretty well, that is, the ideas of: program assigned to a term, and identity criteria for programs or, equivalently, definitional equality between terms. For deepening his understanding he should look at the relevant literature cited in footnote 9 of Note 1. Finally, since in terms of I §3a we are looking here at genuine, terminating programs only, perhaps an even better illustration of (or approximation to) proposed applications would be got from a language in which every term denotes a terminating program, as is the case with typed combinators; for arithmetic problems one adds (typed) recursion operators.[14] Whatever their mathematical appeal (or use in other contexts) the usual combinators do not seem to present any advantages here; specifically the definability of an (even) type free recursion operator does not seem worth the price of introducing expressions for which one does not know whether or not the corresponding program terminates. And--perhaps this is a principal point--it cannot be assumed that the BASIC IDEA OF THE WHOLE PROPOSAL (which comes from the familiar experience of knowing whether or not two familiar definitions denote the same program) EXTENDS UNAMBIGUOUSLY TO THE UNFAMILIAR (full) LANGUAGE

of combinators. So without any prejudice against future theoretical studies on this point, I suggest that the reader concentrate on strongly computable computation diagrams in the sense of I §2(c).

Example. We suppose descriptions of programs to be given by terms built up from typed combinators (of type: $0 \to 0$); cf. footnote 9 of Note 1. Two such terms denote the same program if and only if their (unique) normal forms are congruent, that is, the normal forms are canonical or standard descriptions of programs. Since each term has a normal form, the required decision can be carried out by working out the normal forms.

Corollary (to footnote 12 of Note 1). For practically efficient solutions it will be necessary to find (sub) classes of terms for which an equivalent but computationally simpler criterion of definitional equality can be given. Naturally, as in footnote 14 above, the equivalence proof need not at all be elementary.

3. Discussion of the formalization (proposed in the last section). (a) At first sight the sense of checking seems to be unduly narrow. In fact, a program π_f is, of course, a perfectly good solution even if π_f does not check Φ, but simply satisfies Φ. The justification for the narrow definition, though simple, is often overlooked.

> If, as a matter of empirical fact, programmers tend to write programs which do check the conditions discovered by numerical analysis, it makes good practical sense to exploit this fact.

In this sense the proposal involves an (empirical) hypothesis about programmers which has to be verified.

(b) Note, however, that the notion of definitional equality(on which the formalization is based) is rather wide unless we are planning to use the program for a large number of values of p. It ignores the steps needed to go from the description $t[\pi]$ to its normal form, that is, to the program described, or, equivalently, to its canonical description. It is of course conceivable that an even sharper notion of checking is valid which is based on a narrower notion of definitional equality;

'valid' in the practical sense above of being respected by actual programmers. The discovery of such a notion would probably be at least as useful as the kind of mathematical discovery considered in the Corollary at the end of §2.

(c) Concerning the general hypothesis on the form of functional properties Φ supplied by numerical analysis (according to §1), one can distinguish three kinds.

(i) The equations, read from left to right, translate into strongly computable computation rules; this is the case in the example given: $f(0) \to 1$, $f(p + 1) \to 2f(p)$ (when supplemented by rules for doubling: $2.0 \to 0$, $2(p+1) \to (((2p) + 1) + 1)$).

(ii) Φ, that is, $\forall p \Phi_0(p,f)$ provides a finite[15] definition of f, that is; $\forall m \exists n \exists q([(\forall p \leq q)\Phi_0(p,f)] \to fm = n)$.
In this case f is recursive but the equations in Φ_0 do not necessarily provide a computation diagram.

(iii) Conceivably numerical analysis provides a Φ that defines f uniquely but not computationally, for example,

$$\forall p[f(p) = 2f(p + 1)] .$$

Only $f = \dot{0}$, i.e., $\forall p[f(p) = 0]$ can satisfy this condition since if $f(p) = k$, $f(p + k) = k \cdot 2^{-k}$ which is not integral. But, presumably, the numerical analyst will himself provide this little argument and thereby replace the condition by

$$\forall p[f(p) = 0) .$$

(d) Concerning the existence of genuine uses of the present proposal there can, of course, be no doubt inasmuch as I was able to appeal to such familiar material as exponentiation or doubling (only the range of application is in doubt). For the same reason the weakest point in the illustration is the use of mathematically 'interesting' and therefore unfamiliar material, such as the typed combinators with recursion. No doubt more experienced people would be better able to judge at the present stage.