Fixed versus selfmodifying programs

By J. Nievergelt†

In this note the problem is considered whether or not selfmodifying programs are in principle more powerful than fixed, i.e., nonselfmodifying, ones.

The vague question, "Can programs which are allowed to modify themselves during execution do anything that fixed programs cannot?" can be put in many different ways in a rigorous form, and the answer turns out to be different for different formulations of the problem.

Elgot and Robinson, in a recent paper (1964), find that "in this context it is established that selfmodifying programs are more powerful (can compute more sequential functions) than nonselfmodifying ones, provided that the individual instructions fall within a prescribed, rather broad, class of instructions."

In this note we translate the same vague question into a rigorous one in the framework of Turing machines, and find the opposite answer: for every selfmodifying program there is a fixed one which does the same job (effects the same transformation on the data).

Since both rigorous formulations mentioned seem to us to mirror in a rather natural way the original problem, we conclude that the intuitive question, whether selfmodifying programs are more powerful than fixed ones, cannot be answered with a simple yes or no before it is made more precise.

Statement of the problem

Given:

(1) A universal Turing machine U with initial state q.
(2) A tape T for U which contains only a finite number of non-blank squares.
(3) An arbitrary fixed segment P of the tape.
(4) An initially scanned square s in the segment P.

Question: Does there exist a tape T^* for U with the following properties:

(1) There is a fixed finite segment P^* of T^* such that the outside of P^* on T^* is identical to the outside of P on T.
(2) There is a square s^* in P^* such that, if the computation of U on T beginning with (q, s) terminates, then the computation of U on T^* beginning with (q, s^*) also terminates, and, at termination of the two computations, the outside of P on T is identical to the outside of P^* on T^*.
(3) The computation of U on T^* does not change any symbol of P^*.

This problem admits the following interpretation. U corresponds to the control of a computer, the tape to its memory, which is initially loaded with a program P and data (the "outside of P on T"). The result of the computation is defined to be the expression appearing on the tape outside of P (in the "data area") at termination. The section P will in general be altered during computation, i.e., the program is selfmodifying. We ask for the existence of a program P^* which will perform exactly the same transformation on the data, but which may never be overwritten during execution, i.e., is "fixed".

The answer to the above question is "yes", and we will sketch an explicit procedure for constructing P^* from P. The solution presented is admittedly trivial, but it is based on the important fact that the distinction between program and data is arbitrary. Roughly, the fixed program P^* consists of P and an auxiliary program which first establishes a copy of P in the data area, then proceeds to execute this copy in exactly the same way as the original version of P would have been executed, and finally erases from the data area (the modified form of) this copy of P. Thus the fixed program P^* performs exactly the same transformation on the data as the selfmodifying program P.

Construction of the fixed program

Consider a Turing machine $\bar{U}$ which is an extension of U in the sense that its sets of states, symbols and rules include the corresponding sets of U. $\bar{U}$ will operate on T in such a way that, after some preliminary work, it will imitate exactly the behaviour of U until this latter stops (if it doesn't, neither will $\bar{U}$), at which moment it does some final work and then also stops. More precisely, $\bar{U}$ will perform the following operations:

(1) If the segment P consists of n squares, $\bar{U}$ will begin by moving all the data to the right of P $(n+1)$ squares to the right, and all the data to the left of P one square to the left.
(2) Next, $\bar{U}$ will copy the content of the segment P onto the second to the $(n+1)$th squares to the right of P, and print a special symbol (not contained among the symbols of U) on the two squares immediately to the left and right of P.
(3) Third, $\bar{U}$ will move to that square of the new copy of P which corresponds in its relative position to the initially scanned square s of P, and will enter a state in which it will do everything exactly as U

† *Department of Computer Science, University of Illinois, Urbana, Illinois, U.S.A.*

would, with the only exception that, whenever $\overline{U}$ reads one of the special markers which were printed to delimit P, it will skip across the segment P without altering any symbol on it.

(4) If and when U would stop, $\overline{U}$ will enter a state in which it will perform the inverse of the data-shifting described under (1), i.e., shift all the nonblank symbols to the left of the left marker one square to the right, and all the nonblank symbols at least $n + 2$ squares to the right of $P\,n + 1$ squares to the left, thus overwriting the two special markers and the n squares where the new copy of the program used to be.

Notice that $\overline{U}$ performed on T exactly the same transformation of the data as U would have done, but did not alter the segment P.

The last step now is obvious: since U is universal, we can simulate $\overline{U}$ on it by writing a description of $\overline{U}$ on U's tape. Moreover, this can be (and usually is) done in such a way that this description is never altered. The tape thus constructed is the T^* we have been looking for, with P^* consisting of the description of $\overline{U}$ and the (unalterable) copy of P.

Conclusion

There is an explicit procedure for associating with an arbitrary program, which may modify itself during execution, a fixed (nonselfmodifying) program which performs the same transformation on the data. It has here been described within the framework of Turing machines, but it could obviously be applied to a program for an existing computer, within the limitation of a finite memory, and disregarding the inefficiency of the procedure.

Reference

ELGOT, C. C., and ROBINSON, A. (1964). "Random-Access Stored-Program Machines, an Approach to Programming Languages," *J. Assoc. Comp. Mach.*, Vol. 11, No. 4, p. 365.

Book Review

Numerical Methods: 2. *Differences, Integration and Differential Equations*, by B. Noble, 1964; 372 pages. (Edinburgh: *Oliver and Boyd*, 12s. 6d.)

The author begins with a masterly exposition, in seventeen pages, of the traditional material on finite differences. Operators are avoided in the text but neatly introduced in the exercises. Chebyshev polynomials are also introduced in the opening chapter and the reader is referred for further information, to *Modern Computing Methods*, which is also recommended for the bibliography it offers.

In the next chapter it is pleasant to note the Hartree influence still at work, as evidenced by the classic example on the difficulties of representing a function by a polynomial.

Seven pages are devoted to Aitken's method of interpolation by linear cross-means, terminating with a computer program. A suggestion to the reader to produce a flow-diagram *after* writing the program raises an interesting point. In most non-mathematical problems flow-charts necessarily precede the programs in order not to overlook the many alternative paths involved. In some mathematical problems, such as the one on page 210, it is simpler to write the program in an automatic programming language first. The flow-diagram then becomes valuable as part of the documentation of the program.

Divided differences are included, although not with numerical examples, in order to introduce the usual finite-difference interpolation formulae. Error analysis and modified differences are dealt with and there is a sufficiency of numerical illustration.

The linking of the Trapezium rule and Simpson's rule for numerical integration by means of Richardson's extrapolation formula is neatly demonstrated. The error analysis is thorough, and a computer program is given for Simpson's rule as one of a set of illustrative problems.

Singularities are discussed and Gaussian formulae illustrated by an example.

Most effort of the text on ordinary differential equations is devoted to initial-value problems. Great attention is paid to truncation and round-off error estimation both for a single step and for a number of steps. The chief methods described are those of the Runge-Kutta, Adams-Bashforth and predictor-corrector. Simple illustrations are given to indicate the nature of "ill-conditioning" and instability. The consistency and convergency of finite-difference approximation formulae are also examined.

The author's chapter on partial differential equations, in journalistic terms, can be regarded as "the relations of $\emptyset_{xx}$ with $\emptyset_t$, $\emptyset_{tt}$ and $\emptyset_{yy}$." It covers almost the same field as G. D. Smith's book (*O.U.P.* 1965) in about one-seventh of the space, allowing for page sizes. Smith's book, indeed, would be the natural sequel for anyone wishing to extend his knowledge in this sector.

Similar comparisons could no doubt be made for all the topics covered in both the first volume and the present one. The author has succeeded in giving an up-to-date picture of the state of the art. Newcomers, who study his books, will achieve a perspective to guide them to the more detailed literature. Noble's *Numerical Methods* has shown that miniaturization in "software" can be as successful as that for our computer hardware.

Very few misprints were detected. Programmers may care to debug the program 10.2 on page 285 for a single error. Pages 307 and 308 repeat the omission of a minus sign in $e^{-\alpha X}$.

M. BRIDGER