Short Notes

Editor's Note

The following paper, 'Optimising Self Replicating Programs', would appear on the surface to be a contribution not to be taken too seriously. This is misleading. There is a serious point concerning the need always to analyse fully the logic behind what we are doing. I wish, however, to correct the author on one aspect of the paper; in his references he mentions a page in *The Computer Journal*, vol. 29, no. 6. This particular page is not in the main body of the Journal but is in the Annual Index. The author should note that it is Editorial Board policy not to include in the main body of the Journal pages of the kind to which both he and I refer. Whether or not the reference to vol. 20, no. 4 implies the existence of 'virtual journals' and/or 'virtual papers' I must leave to readers to decide: it is presumably a page fault.

Optimising self-replicating programs

Received January

Introduction

A major and proper concern in computer science is the efficiency of programs, and more generally the weighted efficiency of the combined development cycle and run time. The weighting depends on the expected development-time to life-time ratio of the fully developed program: if it is to be run once, the development time takes on a higher weight; if it can be run many times (or marketed widely) then run time acquires a higher weighting. The conventional wisdom is that judicious use of software tools can drastically reduce the relative cost of software development. This article disputes this position by exhibiting a counter-example.

Definitions

A self-replicating program is defined as a fixed point of the program development cycle. Typically, executing a compiled self-replicating program produces output equal to the program itself. Sometimes, the fixed point is not achieved immediately, but the development sequence must be iterated; in this case we call the program a k-step self-replicating program, where k compile-execute steps are required. In general, replication can occur with non-unit period (i.e. within a non-singleton set of mutually self-replicating programs); however, we shall not explore such possibilities here. *Autonomously* self-replicating programs are termed 'viruses'* and, although the subject of much recreational literature,[1] are a serious security issue.[18]

The existence of self-replicating programs is established by the Kleene recursion theorem.[2,7] Self-replicating programs were first discussed by John von Neumann[17] and caused much discussion, notably the so-called Rosen Paradox.[8] Put briefly, the Rosen Paradox claims that a program certainly cannot be specified until the range of its output is known, but for a self-replicating program the program is in its own range. Thus a self-replicating program cannot be specified. Strachey[11] has given the specification for an impossible program (by taking the domain of the program to include itself),* though Maurer[4] claims no impossibility arises for finite programs.

The proof of the possibility of self-replicating programs assumes an admissible encoding of programs;[3] however, in this paper we are concerned with self-replication without encoding (i.e. identity encoding). It is easily shown that coding is a critical problem, as follows. The General Purpose Macrogenerator (GPM) is clearly Turing complete, so by the recursion theorem we may certainly devise a self-replicating GPM program *up to encoding*. Now, any non-trivial self-replicating GPM program must output the symbol '§' (in order to commence a macro definition or invocation). This symbol must therefore occur in the program in quoted form (e.g. '<§>'). Since the program is self-replicating, occurrence in the input requires occurrence in the output (of the preceding program execution). However, to obtain the necessary quote symbols *they* must be quoted, so the input contains more quotes than the output. Such a program could not be self-replicating. Therefore Turing completeness is an insufficient condition to permit self-replication. In general, a self-replicating programming notation requires not only names for certain objects, but names for program code to compute those names and it is this that the GPM lacks. The lambda calculus supports the self-replicating '$\lambda x(xx)\,\lambda x(xx)$' that reduces to itself without the use of quotation mechanisms, but it should be noted that any well-formed formula of λ-calculus can be the result of reduction, and therefore λ-calculus meets the criterion just mentioned (not only having names for objects but also being able to name expressions reducing to those objects).

Since the GPM is a conventional macro processor non-macro text is processed literally, so non-macro 'programs' self-replicate in a purely trivial sense. The recursion theorem also affirms the possibility of self-replicating universal programs; bearing this in mind will naturally exclude trivial self-replicating programs from further consideration.

Optimising self-replicating programs

Given a k step self-replicating program P, a $k+1$ step self-replicating program can be constructed directly. The $k+1$ step program merely outputs the text of the k step program. If $k > 1$, the $k+1$ step program may take liberties with program layout, since these will be absorbed in the second compilation step. In the present paper we are concerned with optimising self-replicating programs, so we shall search for ways to reduce k. The first example, due to Thomson,[16] is of a 2-step self-replicating program and is shown below.

Despite the somewhat arbitrary appearance of this code, self-replicating programs may be developed systematically from first principles[12].

```
char s[] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
  ... (215 or so lines omitted) ...
        '\n',
        0
};
/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end
 */
main()
{
        int i;
        printf("char\ts[] = {\n");
        for( i = 0; s[i]; i++)
          printf("\t%d, \n", s[i]);
        printf("%s", s);
}
```

The first step in the program towards self-replicating converts the explicit character constants ('\t', '0', etc.) into decimal values. In support of the 'software tools approach' it can be argued that since the second-step self-replicating program is approximately four times longer than the first there is a significant economy to be had in writing the smaller version and using the initial compile-execute step for its expansion. (A slightly longer initial program could have been written to avoid this, though its greater length would encourage the program developer to use a software tool in order to generate the character constant table, thus keeping the number of steps the same!) Without loss of generality we base our analysis on time complexity. Conservative estimates of the development time are: 10 min (600 sec) writing;* 1 min (60 sec) compiling; say, 0.1 sec running (proportional to the length of the program). Since this is a 2-step self-replicating program, achieving the fixed point takes time totalling 720.2 sec, iterated perhaps ten times for debugging (7202 sec), neglecting intermediate editing time.

The following program is an example of a 1-step self-replicating program. (Purely for clarity of presentation it is only 1-step up to white-space encoding – it is in fact typeset below as a 2-step self-replicating program.)

```
char q = '"', *s =
"charq = '%c', *s = %c%s%c; main()\
  { printf(s, q, q, s, q); }";
  main() { printf(s, q, q, s, q); }
```

This program is a third shorter, and similar analysis to that given above shows obtaining self-replication requires 2401 seconds (now *conservatively* allowing for ten iterations for debugging, even though the program is simpler). Clearly, omission of the first software-tools cycle has saved 7202 – 2401 = 4801 sec, a significant 66% saving.

* For example the PDP11-style instruction '`mov 0 1`' when executed makes a copy of itself in the following word, which will then be executed, and so on.

* The fact that Strachey actually exhibits the 'impossible' program means that it is not the program *itself* that is impossible. If the program was run it would crash and there is nothing paradoxical about that. The real issue is that the program fallaciously purports to compute a function that is non-computable.

* Ten minutes when copied from Thomson's paper; it may have taken *him* longer to write.

Nevertheless, we may make greater savings by eliminating the execution step, as follows. The following program is a ½-step self-replicating program:

```
"s.c", line 1: syntax error
```

Compiling this program immediately obtains the fixed point. Eliminating the execution step has obtained a factor of around four in *further* time savings. Notice that in the language C (in which you can see the program above is written) there are no facilities to obtain the name of the program, and it has to be written in explicitly. Similar problems arise in the Prolog 1-step self-replicating program, `'x :- listing(x).'`. Many interpreted languages provide explicit features that permit more general programs, a feature that may be readily exploited in natural language (e.g. Smullyan[9]). For example, in the UNIX* shell we may write `'cat $0'`. In passing, note that `'cat $0; exit;'` can be prefixed to any shell program to cause it (together with the prefix) to self-replicate. The `'exit'` command in the prefix is required to ensure the composite program only self-replicates.† Text formatters (normally directly interpreted) raise further issues for self-replication that I shall address shortly.

The final example, shown below, is a 0-step self-replicating program.

This program does not require writing, compiling nor executing (in short, *no* reliance on software tools). It self-replicates directly. Of course, you have to be careful which software system is not used; for example, in Pascal it causes syntax errors (in C it causes load-time errors). The program works best in Algol-60. As promised in the introduction, this program is efficient without using software tools. Indeed, the program may be used any number of times and yet only replicate itself precisely once.

Self-replicating papers

It is interesting to note that the last program is 2-step self-replicating for most text formatters, since any output (and therefore the least fixed point) is normally a positive multiple of whole pages. We call such a text (program) a 2-step self-replicating paper, and in the general case, a k-step self-replicating paper. A 1-step self-replicating paper may be found in Thimbleby,[14] and we note that there is a significant literature of k-step ($k > 1$) self-replicating papers with which the reader is probably already well enough acquainted. Self-replicating papers should be carefully distinguished from self-referencing[5,15] and self-satisfied (tautologous) papers.

A small conjecture

I finish this paper offering a conjecture: non-trivial, non-encoded* self-replicating papers cannot be written in **nroff**. If proved, this would undermine Reid's otherwise valid criticism of **nroff**.[6]

H. THIMBLEBY
Department of Computer Science,
University of York,
York YO1 5DD

References

1. A. K. Dewdney, Computer recreations. *Scientific American* **250** (5), 15–19; **252** (3), 14–19 (1984, 1985).
2. S. C. Kleene, *Introduction to Metamathmatics*. Van Nostrand, Reinhold (1952).
3. C. Y. Lee, A Turing machine which prints its own code script. *Proceedings of the Symposium on Mathematical Theory of Automata*, pp. 155–164. Polytechnic Press, Polytechnic Institute of Brooklyn (1962).
4. W. D. Maurer, Correspondence. *Computer Journal*, **8** (4), 330 (1966).
5. J. Morton, On recursive reference. *Cognition* **4** (4), 309 (1976).
6. B. K. Reid, The Scribe document specification language and its compiler. *International Conference on Research and Trends in Document Preparation Systems*, pp. 59–62. Swiss Institutes of Technology, Lausanne (1981).
7. H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York (1967).
8. R. Rosen, On a logical paradox implicit in the notion of self-reproducing automata. *Bulletin of Mathematical Biophysics* **21**, 387 (1959).
9. R. M. Smullyan, *What Is The Name Of This Book*? Prentice-Hall, Englewood Cliffs, NJ. (1978).
10. C. Strachey, A general purpose macrogenerator. *Computer Journal* **8** (3), 225–241 (1966).
11. C. Strachey, An impossible program. *Computer Journal* 7 (4), 313 (1965).
12. J. W. Thatcher, The Construction of a self-describing Turing machine. *Proceedings of the Symposium on Mathematical Theory of Automata*, pp. 165–171. Polytechnic Press, Polytechnic Institute of Brooklyn (1962).
13. H. W. Thimbleby, *Computer Journal*, **20** (4), 386–400 (1977).
14. H. W. Thimbleby,* *Computer Journal*, **29** (6), x (1986).
15. H. W. Thimbleby, Optimising Self-Replicating Programs. *Computer Journal*, this issue.
16. K. Thomson, Reflections on trusting trust. *Communications ACM* **27** (8), 761–763 (1984).
17. J. von Neumann, The general and logical theory of automata. In *Cerebral Mechanisms in Behaviour. Proceedings of 1948 Hixon Symposium*, edited L. A. Jeffress, pp. 1–41. Wiley, New York (1951).
18. I. H. Witten, Computer (in)security (Infiltrating open systems). Department of Computer Science, Research Report no. 86/249/23, University of Calgary (1986).

Editor's Note (author's version)

In order to ensure the timely publication of this important paper, I have taken the very unusual step of publishing it together with referee's comments.

'The paper "Optimising self-replicating programs" by H. Thimbleby addresses an important issue that deserves wider appreciation. The paper is both good and original, but unfortunately the good parts are not original and the original parts are not good (cf. S. Gorn's *Compendium of Rarely Used Clichés*). This should not be seen to detract from the basic argument, for the same observation applies to the previous sentence also. I recommend the paper be published promptly, but only once.'

* UNIX is both a trademark and footnote of AT & T Bell Laboratories.

† Note that the shell examples are only weakly self-replicating (i.e. only if they terminate successfully will they have self-replicated). If the program code is located by pathname search, then `'$0'` is not necessarily the name of the program code file. To overcome this problem, the program could be written, `'cat `which $0` '`, though there are still the possibilities that the output from `which` is not unique, and that the program may not have been invoked from the shell (in which case `'$0'` is arbitrary).

* Non-encoded requires not changing quote/escape symbols nor using the transliterate (`'.tr'`) command. Encoded self-replicating papers in **nroff** are trivial, as the earlier discussion on self-replication in GPM intimated.

* The actual printed paper is anonymous, but I can now admit I submitted it as a 2-step self-replicating paper (the version printed is one page longer than my original). Owing to an unexplained time anomaly, a corrected version was printed earlier.[13]