To demonstrate that a set of clauses is inconsistent, using the matrix connection method, we simply show that each path through the set is closed by the presence of at least one complementary pair of literals. In the example above, path (*a*) is closed by the pair $\neg P$, P, path (*b*) is closed by the pair Q, $\neg Q$, and path (*c*) is closed by the pair R, $\neg R$. Hence we have shown that the set of clauses is inconsistent.

To prove that a formula is a theorem of a set of formulas, we proceed as follows: (*a*) convert the set of formulas to a clause set S; (*b*) negate the formula being tested for theoremhood and convert to a set of clauses C; (*c*) check the clause set $S \cup C$ for consistency as described above; (*d*) if $S \cup C$ is inconsistent then we have succeeded in our proof.

The problem which we have identified concerns the analysis of algorithms which have been developed to implement the matrix connection method. Although intuitively appealing algorithms have been developed, for example by Wallen at Edinburgh University for the general case, and by Frost at Glasgow University for the propositional case, the mathematical machinery to analyse the performance of such algorithms has not been fully developed.

A domain-independent description of the 'same' problem

The problem above may be re-expressed as the problem of determining the efficiency of algorithms which check for the existence or lack of existence of an open (unclosed) path in an array of integers such as the following:

$$\begin{array}{lll} -1 & 2 & 3 \\ -2 & & \\ -3 & & \\ 1 & & \end{array}$$

where 'path' and 'closed', in this case, are defined analogously to the definitions given earlier.

Essentially, we want the mathematical machinery which will allow us (*a*) to describe various types of array (e.g. types in which 75% of rows contain a single integer), (*b*) to describe various algorithms based on different heuristics (e.g. algorithms which delete rows which contain an integer whose negation does not appear elsewhere), (*c*) to determine the efficiency of various algorithms in different cases, and (*d*) to identify exact bounds on the computational complexity of various cases. The ultimate aim is to identify the provably best algorithms for consistency checking in various cases. When we first described the problem in the domain-independent terms above, we had the feeling that this problem must have cropped up elsewhere, possibly in networking or scheduling applications, and that the mathematical machinery which we required had, most likely, been developed elsewhere. Consequently, we approached various people working on networking/scheduling problems in operational research and mathematics departments. Unfortunately, we have not yet found a ready-made solution, and have begun to develop our own techniques. However, we are not pursuing this work with any great urgency since we are hoping that this short note will prompt someone to write to us saying that they recognize the problem and can direct us to a solution.

(On reading through the draft of these notes, we realised that we had not followed suggestion (*b*) when we chose a title. Our solution was to put parentheses round the domain-specific part of the title).

R. A. FROST
Department of Computing Science,
The University of Glasgow,
Glasgow G12 8QQ.

References

1. A. Deliyanni and R. A. Kowalski, Logic and semantic networks. *CACM* **22** (3), 184–192 (1979).
2. D. A. Prawitz, A proof procedure with matrix reduction. In *Lecture Notes in Mathematics* 125, pp. 207–213. Springer, Berlin (1976).
3. P. B. Andrews, Theorem proving via general matings. *JACM* **28**, 193–214 (1981).
4. W. Bibel, Matings in matrices. *CACM* **26** (11), 844–852 (1983).

Writing Self-replicating Code

Writing a nontrivial self-replicating program is not difficult, yet when encountered with the task people in general seem confused. In this note we argue that this need not be the case.

Received July 1986

1. Introduction

Writing self-replicating code is fun and has educational value but it seems that people in general find it confusing. In this note we argue that this can be overcome and give a Franz Lisp function which does the job. The function is sufficiently general to let the reader solve the problem in another language. Our impression is that the problem is also a nice exercise to see the real power of your favourite language 'printability'.

Let us first define what we mean by self-replicating code.

Let P be a source program which resides in file F. Then P is self-replicating if, when executed, it writes an exact copy of the contents of F to e.g. the terminal.

Several points need clarification in this informal definition. First, it may or may not be the case that P needs compilation. For instance, in our case P will be in Franz Lisp and be interpreted (although it also works under compilation). In another language, e.g. Fortran, the assumption is that you first compile your program. Second, P must be in a source language. This is required just to make the exercise interesting. If one allows a 'program' written, for example, in binary (a string of ones and zeros), then Lisp, for instance, will simply echo the string back on your terminal and there you get the answer! In fact, this furthermore proves that any binary program (be it self-replicating or not) is self-replicating from Lisp's viewpoint. Clearly, such a program is not very interesting. In a similar way, there exist two short programs in Lisp (namely *t* and *nil*) which are also self-replicating but dull.* The triviality of the preceding examples are due to Lisp's insistence in reading, evaluating, and printing – a fact often referred to as a *read–eval–print* loop. Thus one is normally expected to write something less trivial. That is what we are going to do in the sequel.

2. The program

The complete program is given in Fig. 1. (The line numbers inside square brackets are for identification purposes only and are no part of the code.) It consists of a function *r* which, when executed by Franz Lisp (Opus 36), writes a copy of itself on the terminal. We assume familiarity with this particular dialect of Lisp. (The reader is referred to the Franz Lisp Manual[1] and Wilensky's textbook[2] for details.) The function is not of minimal size nor is it given in its nicest looking (from Lisp's viewpoint) form.

To start with, it is obvious that one should store some information about the program itself, for example, an array. Here *a* serves this purpose. (Franz indexes arrays starting at 0, we do not use the 0th entry of *a*.) Thus, lines 3 and 4 simply store the 'declarative' statements in the beginning of the function. Then by storing the printing functions for these lines in *a* we make sure that they will be generated in the replication (lines 5 and 6). The core of the replicative process is using the values stored in *a* in a careful way (i.e. the *do*-loop in line 7). The statement which starts at line 8 and ends at line 13 stores the program segment given in lines 14 to 19 in *a*. (The careful reader will note that the very *nil* at the end of the program is there just to get around the fact that Lisp will return *nil* when *r* is

```
 [1] (defun r () (prog (i)
 [2] (array a t 7)
 [3] (store (a 1) "(defun r () (prog (i)")
 [4] (store (a 2) "(array a t 7)")
 [5] (store (a 3) "(princ (a 1)) (terpr)")
 [6] (store (a 4) "(princ (a 2)) (terpr)")
 [7] (store (a 5) "(do i 1 (1+i) (> i 5)")
 [8] (store (a 6) '|(princ (a 1)) (terpr) (princ
     (a 2)) (terpr)
 [9] (do i 1 (1+i) (> i 5)
[10] (princ "(store (a') (print i) (princ ")")
     (print (a i)) (princ ")")
[11] (terpr))
[12] (princ "(store (a 6)'") (print (a 6)) (princ
     ")") (terpr)
[13] (princ (a 6)) (return)))|)
[14] (princ (a 1)) (terpr) (princ (a 2)) (terpr)
[15] (do i 1 (1+i) (> i 5)
[16] (princ "(store (a") (print i) (princ ")")
     (print (a i)) (princ ")")
[17] (terpr))
[18] (princ "(store (a 6)'") (print (a 6)) (princ
     ")") (terpr)
[19] (princ (a 6)) (return)))nil
```

Figure 1

* It may safely be said that Lisp supports the shortest self-replicating program!

executed. Thus in our case *F* consists of *r* followed by the program *nil*.) In general, the most important issue during the design was to solve the printing difficulties. For instance, the reader will note the difference between the value stored in (*a*6) and other entries of *a*. In (*a*6) we cannot use a string as in the previous cases since the value we are trying to store has strings already. Thus the use of | | is invoked. The reader should carefully check *r* to make sure that all details are understood.

3. Discussion

It is trivial to modify *r* to make it recursive. Then one would have a program which replicates itself on a terminal indefinitely.

Although we have not taken the direction, any optimization version will also be interesting. In this respect, a better problem may be the following:

Write a self-replicating *P* such that it does something useful and is of small 'size' (total number of characters in *F*). (Here the notion of being useful is left to the imagination.)

We invite interested readers to send us copies of their favourite self-replicating programs written in a familiar language. These will be included (with proper credits) in a future anthology we are planning on the subject.

Acknowledgement

I thank Matthijs Kuiper for suggesting this problem to me.

V. AKMAN
Department of Computer Science, University of Utrecht, Budapestlaan 6, P.O. Box 80.012
3508 TA Utrecht, the Netherlands

References

1. J. K. Foderaro and K. L. Sklower, *The FRANZ LISP Manual.* University of California, Berkeley, Calif. (1981).
2. R. Wilensky, *LISPcraft. Norton, New York* (1984).

Correspondence

Algorithms for the Even Distribution of Entities

Dear Sir,

In recent editions of *The Computer Journal* you have published correspondence on algorithms for the even distribution of entities (Compton, Brokate, Elston). In order that readers unfamiliar with the subject should not think this an undeveloped field, the following is a brief review of the major achievements already extant. Many of these works have been published in, or referenced by articles in *The Computer Journal.*

The problem has a particular relevance to computer graphics. This is because drawing straight lines with devices which use the logic of incremental plotters or raster scan tubes requires the even distribution of axial and diagonal moves. All lines that can be drawn must consist of a sequence of such movements, and the straightness of the line depends upon the even distribution of the move types. The sequence of such movements, which represent a straight line, is called its chain code (Freeman, 1970).

In 1965 J. E. Bresenham produced an entirely integer-based algorithm which correctly computed the appropriate move distribution, using nothing more than sign tests and additions. It is still the most regularly implemented line-drawing algorithm, and because of its seminal importance, it is reproduced in the following Pascal formulation.

```
PROGRAM Bresenham (INPUT,OUTPUT)
*constructs the series of 0's and 1's which produce the*
*best-fit from the origin to (u,v) with u>v)*
VAR u,v,a,b,d,counter: INTEGER;

BEGIN
  READLN(u,v);
  b:=2*v;
  a:=2*u-2*v;
  d:=2*v-u;
  counter:=0;
  WHILE counter<u DO
  BEGIN
    IF d<0 THEN BEGIN
                  WRITE('0');
                  d:=d+b;
                END;
          ELSE BEGIN
                  WRITE('1');
                  d:=d-a;
                END;
    counter:=SUCC(counter);
  END;
END.
```

The algorithm resolves the equal-error anomaly by selecting an upper grid point (thus the line from the origin to (2,1) is 10 and not 01). It generates the 'best-fit' chain code specified by minimising either the vertical or perpendicular error distance (Bresenham, 1963). Bresenham's algorithm has been modified to generate circles and conics (Bresenham, 1977; Pitteway, 1967).

Brons (1974) produced an algorithm for generating chain code by using continued fraction expansion via the division version of Euclid's algorithm (a more sophisticated method than quoted in recent correspondence). This was modified by Archelli and Massarotti (1978) to generate 'best-fit' chain code. Brons (1974) has also produced a grammar for the highly context-sensitive language which encodes the even distribution of entities in one dimension.

Modifications have been made to Bresenham's algorithm to improve its operational efficiency (Pitteway and Green, 1982), and to generate reversible plotter paths (Boothroyd and Hamilton, 1970). More recently, Castle and Pitteway (1985) have produced an algorithm which generates 'best-fit' chain code from the palindromic symmetry inherent in its patterns.

The converse problem, that of recognising chain code as belonging to a particular straight line, has been addressed by Dorst & Duin (1984). They have developed an application of spirographic methods to act as a recogniser and estimator for the accuracy of chain code representations.

The purpose of this note is to describe the solutions to this problem that have already been produced, in order to encourage further novel contributions. It is also interesting to reflect that Compton's original problem was unrelated to the field of computer graphics, but is an application of work already done in that area.

References

1. C. Archelli and A. Massarotti, On the parallel generation of straight digital lines. *Computer Graphics and Image Processing* **7** (1), 67–83 (1978).
2. J. E. Bresenham, An incremental algorithm for digital plotting. *Proc. ACM National Conference* (1963).
3. J. E. Bresenham, Algorithm for computer control of a digital plotter. *IBM Systems Journal*, **4** (1), 25–30 (1965).
4. J. E. Bresenham, A linear algorithm for incremental display of digital arcs. *ACM Communications* **20**, 100–106 (1977).
5. J. Boothroyd and P. A. Hamilton, Exactly reversible plotter paths. *Australian Computer Journal* **2** (1), 20–21 (1970).
6. R. Brons, Linguistic methods for the description of straight lines upon a grid. *Computer Graphics and Image Processing* **3** (1974).
7. C. M. Castle and M. L. V. Pitteway, An application of Euclid's algorithm to drawing straight lines. *Proc. NATO ASI on Fundamental Algorithms for Computer graphic*, pp. 135–140. (1985). Springer-Verlag.
8. C. M. Castle and M. L. V. Pitteway, An efficient structural technique for encoding 'best-fit' straight lines. *The Computer Journal.* (Accepted for publication.)
9. L. Dorst & R. Duin, A framework for calculations on digitised straight lines. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, **PAMI–6**, (5) (1984).
10. H. Freeman, Boundary encoding and processing. In *Picture Processing and Psychopictorics*, pp. 241–266. Academic Press, New York (1970).
11. M. L. V. Pitteway, Algorithm for drawing ellipses or hyperbolae with a digital plotter. *The Computer Journal* **10**, 282–289 (1967).
12. M. L. V. Pitteway and A. Green, Bresenham's algorithm with run-line coding shortcut. *The Computer Journal* **25** (1), 114–115 (1982).

Yours faithfully

C. M. CASTLE
Head of Department of Computing and Information Technology, St Mary's College, Twickenham TW1 4SX
M. L. V. PITTEWAY
Professor of Computer Science, Brunel University, Uxbridge UB8 3PH