Speeding up programs

R. Bird

Department of Computer Science, University of Reading, Whiteknights Park, Reading, Berkshire RG6 2AF

A simple programming language L is described with the property that each program in L can be transformed into an equivalent program in L which executes only half as many instructions. The transformation is illustrated by speeding up an example program.

(Received May 1973)

1. Introduction

Imagine a programming language L with the following desirable property: every program in L can be systematically transformed into another program, also in L and equivalent to the first, but which executes only half as many instructions. Programming in L has the pleasant consequence that any program can be made to run faster by an arbitrary linear factor. Of course, a more precise statement of the property, which we shall call *speed-up*, depends on exactly what constitutes an atomic instruction in L.

It is a surprising fact that non-trivial programming languages (i.e. languages capable of computing every recursive function) possessing speed-up can be constructed. The well known language of Turing machines is one example (see Hartmanis and Stearns, 1965), but in some respects not a satisfactory one. The language bears little relationship to what one normally means by a programming language, and the speed-up property only holds because Turing machines are not restricted as to the size of their alphabet of working symbols. In other words, the underlying hardware can be changed. Unfortunately, the speed-up property does not hold for the language of Turing machines which work over a fixed alphabet.

There are, however, more natural examples. It is the purpose of this paper to describe one such language, which has certain features in common with machine code, and demonstrate that the speed-up property holds.

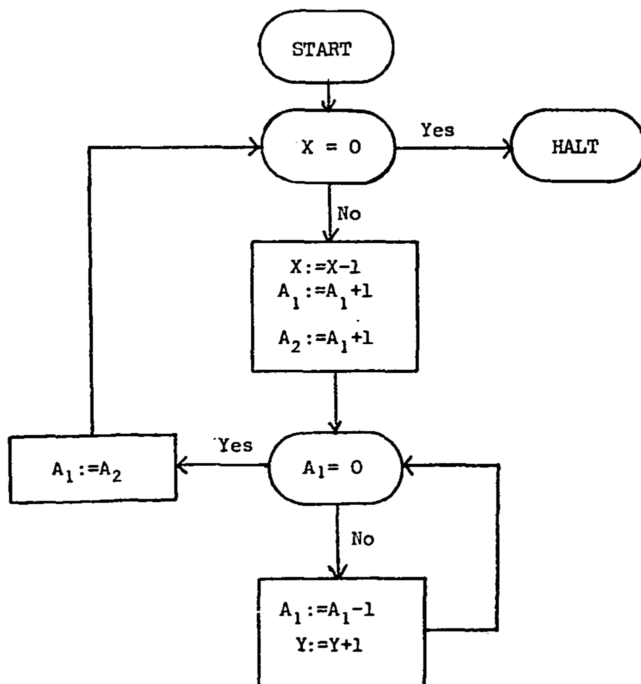

Fig. 1

2. A simple language

Consider the program P of **Fig. 1.** P is a program in the language L which consists of arbitrary flowcharts defined over the following types of instruction:

assignments: $A_i := A_j + 1$, $A_i := A_j - 1$, $A_i := A_j$

tests: $A_i = 0?$

input-output instructions: $X := X - 1$, $Y := Y + 1$, $X = 0?$

where $i, j \geqslant 1$. If P is run with the contents of the input register X initialised to some non-negative integer x, and all other registers set to zero, then P will eventually terminate with x^2 in the output register Y. The form of the input and output instructions effectively restrict X to be a read-only register, and Y to be a write-only register. The reason for this restriction is that in any speeded-up version of P it must still take x^2 instructions of the form $Y := Y + 1$ to store the answer, and $2x + 1$ instructions of the form $X := X - 1$ and $X = 0?$ to read the input. Since this constant overhead can never be avoided, it is simpler not to count instructions involving X and Y as contributing towards the running time. This assumption is then balanced by the consideration that such instructions can only read the input and store the output. With this provision, L can be shown to possess the speed-up property.

The machine M on which program P is run, is essentially the URM of Shepherdson and Sturgis (1963), and consists of the special registers X and Y and an infinite number of work registers $A_1, A_2, \ldots$, each of which can contain an arbitrary non-negative integer. The interpretations of the instructions are the obvious ones, except that an instruction $A_i := A_j - 1$ executed when the contents of A_j are zero, will be supposed to set A_i to zero. The input function of M loads an integer in X, and sets all other registers to zero. The output function extracts the final value of Y.

Not all these features are important for speed-up, however. The particular form of L and M given here has the advantage of simplicity, but we could have allowed:

(*a*) an arbitrary number of input registers $X_1, X_2, \ldots$, so that functions of more than one argument can be computed,

(*b*) arbitrary integers, positive or negative, to be stored in the work registers, and supplemented the instruction repertoire with the extra tests $A_i \geqslant 0$ and $A_i > 0$, etc.

(*c*) only integers below a certain size to be stored in the work registers, to make M more like real computers,

(*d*) certain other simple operations in L, such as the assignment $A_i := 0$.

In each case, the proof requires only slight modification.

Each program P in L computes a partial number theoretic function of one argument, which we denote by fP. Two programs P and Q are *equivalent* if $fP = fQ$, i.e. they compute exactly the same partial function. The *running-time* tP of a

Downloaded from https://academic.oup.com/comjnl/article/17/4/337/443710 by guest on 19 April 2024

program P is defined for each x only if the computation of P with input x terminates, in which case

$tP(x)$ = the number of work-register instructions executed by P when run on input x.

Thus for the program P of Fig. 1,

$$fP(x) = x^2$$
$$tP(x) = 2x^2 + 4x \ .$$

The main result can now be stated formally.

Speed-up property:

For each program P, an equivalent program Q can be found such that $tQ(x) \leqslant tP(x)/2$ for all x.

3. Proof of speed-up property

The running time of a program P is the sum of two functions aP and bP, where

$aP(x)$ = the number of assignments executed by P, when run on input x.

$bP(x)$ = similarly, the number of tests.

We can always modify a given program P to ensure that in every execution of P, no test $A_i = 0$ is ever obeyed twice without at least one assignment to A_i being obeyed in between. Furthermore, because the input convention initialises each A_i to zero, we can arrange that no test $A_i = 0$ occurs before the first assignment to A_i. These observations guarantee the truth of the following lemma.

Lemma:

For each program P, an equivalent program Q can be found such that

$$tQ(x) \leqslant 2aP(x)$$

for all x.

This result shows that to prove speed-up it is sufficient to concentrate attention on reducing assignment instructions, i.e. to prove Theorem 1.

Theorem 1:

For each program P an equivalent program Q can be found such that

$$aQ(x) \leqslant aP(x)/2$$

for all x.

If Theorem 1 is applied twice to a given program P, a program Q is obtained for which $aQ \leqslant aP/4$. By the lemma, we can then find a program R such that

$$tR \leqslant 2aQ \leqslant aP/2 \leqslant tP/2 \ ,$$

and so speed-up is assured.

Rather than give a formal proof of Theorem 1, we shall indicate the general method by speeding up the program P of Fig. 1. The translation to the final program is broken down into a number of stages. The first step is to write P as the following set of labelled instructions:

$1: X = 0 \rightarrow 0, 2$	$5: A = 0 \rightarrow 8, 6$
$2: X := X - 1 \rightarrow 3$	$6: A := A - 1 \rightarrow 7$
$3: A := A + 1 \rightarrow 4$	$7: Y := Y + 1 \rightarrow 5$
$4: B := A + 1 \rightarrow 5$	$8: A := B \rightarrow 1$

where to avoid subscripts, registers A_1 and A_2 have been renamed A and B. An instruction such as $A = 0 \rightarrow 8, 6$ is equivalent to the ALGOL statement **if** $A = 0$ **then goto** 8 **else goto** 6. Termination occurs when label 0 is reached.

The second step is to convert P into a program Q which executes half as many assignments, but which makes use of the more general instruction types

$$A_i := A_j + d, A_i := A_j - d, A_i = d? \text{ for some } d \geqslant 0 \ .$$

At a later stage these instructions will be replaced by the legally allowed set.

Roughly speaking, Q simulates P in a step by step manner, except it delays the execution of assignment statements. Each such instruction executed by P is saved in the label structure of Q until a sequence of sufficient length has been built up to enable an equivalent sequence (over the extended instruction types) of no more than half the length of the original to be defined. Q then executes this equivalent sequence. It is, of course, important to know that an equivalent sequence with the desired property can always be found. For our example, any sequence of 4 instructions from the set

$$A := A + 1, B := A + 1, A := A - 1, A := B$$

can be replaced by an equivalent sequence of length 2 over a suitably extended set. This follows as the special case $n = 2$ of the following general result.

Theorem 2:

Let L be a finite sequence of assignments of the form $A_i := A_j + d$, $A_i := A_j - d$, where $d \geqslant 0$, and $1 \leqslant i, j \leqslant n$.

It is possible to construct a similar sequence L', equivalent to L such that

$$|L'| \leqslant \min\left(|L|, n + \left[\frac{n+1}{2}\right] - 1\right),$$

where $|L|$ denotes the length of L. In particular, if $|L| = 3n$ (or 4, if $n = 2$), then $|L'| \leqslant |L|/2$.

The proof of Theorem 2, which is straightforward but rather long, is given in Bird (1973).

The first two instructions of Q are

$$1: X = 0 \rightarrow 0, 2$$

and

$$2: X := X - 1 \rightarrow 3$$

since no assignments have yet to be remembered. The next two instructions are the unconditional jumps

$$3: \rightarrow (a, 4)$$

and

$$(a, 4): \rightarrow (ab, 5) \ ,$$

where a is used as shorthand for $A := A + 1$, and b as shorthand for $B := A + 1$. At the label $(ab, 5)$, for example, Q is remembering the sequence $A := A + 1; B := A + 1$ for later execution. These unconditional jumps will later be eliminated.

The next instruction is

$$(ab, 5): \rightarrow (ab, 6) \ ,$$

since it can be determined from the remembered sequence ab, that A cannot be zero at this point. Continuing, we define

$$(ab, 6): \rightarrow (abc, 7)$$
$$(abc, 7): Y := Y + 1 \rightarrow (abc, 5)$$
$$(abc, 5): A = 0 \rightarrow (abc, 8), (abc, 6) \ ,$$

where c is shorthand for $A := A - 1$. The effect of the sequence abc is to leave A unchanged, so the test $A = 0$ must be performed. Since the sequence $abcc$ is equivalent to the sequence $B := A + 2; A := A - 1$, the next instructions are

$$(abc, 6): B := A + 2 \rightarrow l_1$$
$$l_1: A := A - 1 \rightarrow 7$$
$$7: Y := Y + 1 \rightarrow 5 \ ,$$

where l_1 is some new label. Continuing in this fashion, the rest of Q is found to be

$(abc, 8): B := A + 2 \rightarrow l_2$	$(cda, 4): A := B + 1 \rightarrow l_4$
$l_2: A := B \rightarrow 1$	$l_4: B := B + 2 \rightarrow 5$
$5: A = 0 \rightarrow 8, 6$	$8: \rightarrow (d, 1)$
$6: \rightarrow (c, 7)$	$(d, 1): X = 0 \rightarrow (d, 0), (d, 2)$
$(c, 7): Y: = Y + 1 \rightarrow (c, 5)$	$(d, 2): X := X - 1 \rightarrow (d, 3)$
$(c, 5): A = 1 \rightarrow (c, 8), (c, 6)$	$(d, 3): \rightarrow (da, 4)$
$(c, 6): A := A - 2 \rightarrow 7$	$(da, 4): \rightarrow (dab, 5)$
$(c, 8): \rightarrow (cd, 1)$	$(dab, 5): \rightarrow (dab, 6)$
$(cd, 1): X = 0 \rightarrow (cd, 0), (cd, 2)$	$(dab, 6): A := B \rightarrow l_3$

Downloaded from https://academic.oup.com/comjnl/article/17/4/337/443710 by guest on 19 April 2024

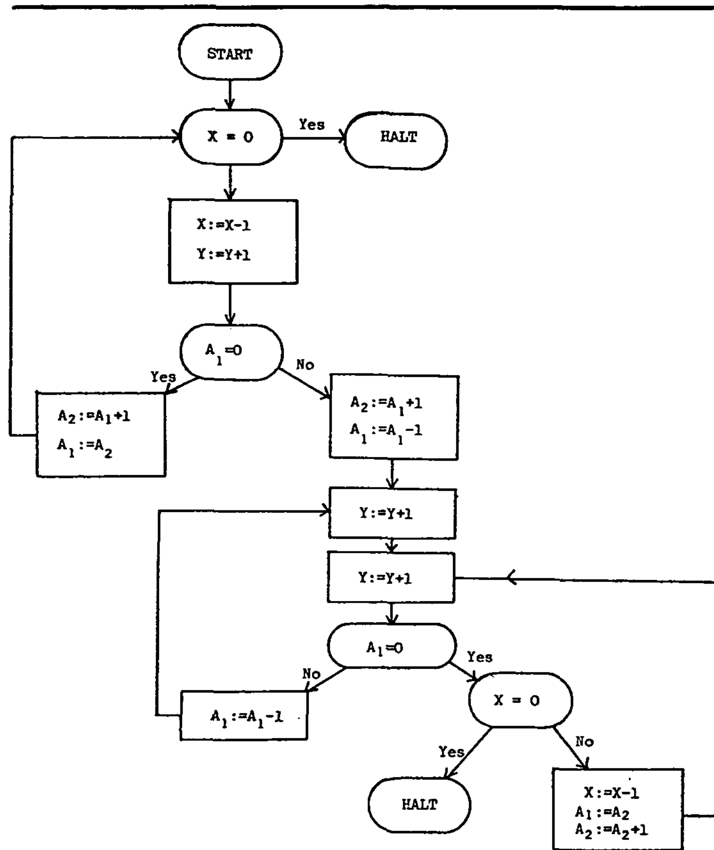

Fig. 2

$(cd, 2): X := X - 1 \rightarrow (cd, 3) \quad l_3: B := A + 1 \rightarrow 7,$
$(cd, 3): \rightarrow (cda, 4)$

where d is shorthand for $A := B$. The unconditional jumps can now be eliminated, leaving the following 21 instructions for Q:

1: $X = 0 \rightarrow 0, 2$	12: $B := A + 2 \rightarrow 13$
2: $X := X - 1 \rightarrow 3$	13: $A := B \rightarrow 1$
3: $Y := Y + 1 \rightarrow 4$	14: $X = 0 \rightarrow 0, 15$
4: $A = 0 \rightarrow 12, 5$	15: $X := X - 1 \rightarrow 16$
5: $B := A + 2 \rightarrow 6$	16: $A := B \rightarrow 17$
6: $A := A - 1 \rightarrow 7$	17: $B := A + 1 \rightarrow 7$
7: $Y := Y + 1 \rightarrow 8$	18: $X = 0 \rightarrow 0, 19$
8: $A = 0 \rightarrow 14, 9$	19: $X := X - 1 \rightarrow 20$
9: $Y := Y + 1 \rightarrow 10$	20: $A := B + 1 \rightarrow 21$
10: $A = 1 \rightarrow 18, 11$	21: $B := B + 2 \rightarrow 8$
11: $A := A - 2 \rightarrow 7$	

At this point, we know that Q is equivalent to P and $aQ \leqslant aP/2$. On the other hand, Q makes use of the extra instructions

$$B := A + 2,\ A := A - 2,\ B := B + 2,\ A = 1\,?\ ,$$

which must be eliminated. This is achieved by converting Q into a program R, which simulates Q, but which stores only half the contents of the registers. Each label in R is of the form

$$(\alpha\beta, l)$$

where l is a label of Q, and $0 \leqslant \alpha, \beta \leqslant 1$. The guiding principle behind the design of R is that if at some stage during an execution of R, a label $(\alpha\beta, l)$ is reached with contents (a, b) of the registers A and B, then at the same stage during the corresponding execution of Q, the label l will be reached with contents $(2a + \alpha, 2b + \beta)$ of the registers.

The first instruction of R is

$$(00, 1): X = 0 \rightarrow (00, 0), (00, 2)\ ,$$

since by the input convention, both A and B are initialised to zero. The other instructions are

$(00, 2):\ X := X - 1 \rightarrow (00, 3)$
$(00, 3):\ Y := Y + 1 \rightarrow (00, 4)$
$(00, 4): A = 0 \rightarrow (00, 12), (00, 5)$
$(00, 5): B := A + 1 \rightarrow (00, 6)$
$(00, 6): A := A - 1 \rightarrow (10, 7)$
$(10, 7):\ Y := Y + 1 \rightarrow (10, 8)$
$(10, 8): \rightarrow (10, 9)$
$(10, 9):\ Y := Y + 1 \rightarrow (10, 10)$
$(10, 10): A = 0 \rightarrow (10, 18), (10, 11)$
$(10, 11): A := A - 1 \rightarrow (10, 7)$
$(00, 12): B := A + 1 \rightarrow (00, 13)$
$(00, 13): A := B \rightarrow (00, 1)$
$(10, 18): X = 0 \rightarrow (10, 0), (10, 19)$
$(10, 19): X := X - 1 \rightarrow (10, 20)$
$(10, 20): A := B \rightarrow (10, 21)$
$(10, 21): B := B + 1 \rightarrow (10, 8)\ .$

The program R is given as a flowchart in **Fig. 2.** R satisfies the conditions of Theorem 1, since R is equivalent to P and $aR \leqslant aP/2$. Actually, R has a running time given by

$$\begin{aligned} tR(x) &= x^2 + 2x - 1 && \text{for } x \leqslant 2 \\ &= 0 && \text{for } x = 0 \\ &= 3 && \text{for } x = 1, \end{aligned}$$

and so $tR \leqslant tP/2$. In this particular case, the lemma does not have to be invoked.

References

Bird, R. S. (1974). Languages with Speed-up, *University of Reading, Department of Computer Science Technical Report* (in preparation)
Hartmanis, J., and Stearns, H. E. (1965). On the computational complexity of algorithms, *Trans. Amer. Math. Soc.* pp. 285-306.
Shepherdson, J. C. and Sturgis, H. E. (1963). Computability of recursive functions, *JACM*, pp. 217-255.

Downloaded from https://academic.oup.com/comjnl/article/17/4/337/443710 by guest on 19 April 2024