Acta Informatica 18, 181 – 206 (1982)

Deriving very Efficient Algorithms for Evaluating Linear Recurrence Relations Using the Program Transformation Technique

A. Pettorossi [1] and R.M. Burstall [2]

[1] IASI – CNR, Via Buonarroti 12, Roma, Italy
[2] Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland

Summary. Using the program transformation technique we derive some algorithms for evaluating linear recurrence relations in logarithmic time. The particular case of the Fibonacci function is first considered and a comparison with the conventional matrix exponentiation algorithm is made. This comparison allows us also to contrast the transformation technique and the stepwise refinement technique underlining some interesting features of the former one. Through the examples given we also explain why those features are interesting for a useful and reliable program construction methodology.

1. Introduction

Transforming recursive equations is a good methodology for writing correct and efficient programs [8]. Following this methodology the programmer is first asked to be concerned with program correctness only and then, at later stages, with efficiency considerations. The original version of the program, which he may easily prove correct, is transformed (perhaps in several steps) into a program which is still correct, because the rules used for the transformation preserve correctness, and it is more efficient because the evoked computations save time and/or space. Several papers have been written about this methodology concerning: (i) *systems* for transforming programs [5, 13, 14, 16], (ii) various *rules* for making such transformations [1, 10, 25, 26, 28, 32], and (iii) some *theories* for proving the correctness of the transformations [4, 20, 21]. The given lists of references are not to be considered as exhaustive. For a more extensive bibliography one may refer to [9, 24].

Unfortunately for the program transformation methodology a general framework in which one can prove that transformations improve efficiency while preserving correctness, is not fully available yet. A first step in this direction was done in [33].

One might wonder in fact whether it is important to develop such a general framework at all: it might be the case that using the transformation methodology

0001-5903/82/0018/0181/$05.20

one can derive algorithms which have a limited degree of efficiency only, and that if one wanted to write very efficient programs one would be forced to adopt other programming techniques, as for example, the stepwise refinement technique [12, 35]. But this does not seem to be the case, at least for particular classes of algorithms, as demonstrated in this paper for algorithms which evaluate linear recurrence relations, such as the well-known Fibonacci one. Working out the details of the program transformation, we show that very high levels of efficiency can be achieved.

This problem was suggested to us by Professor Dijkstra, who considered it to be a challenging one for exploring the practical interest of the program transformation methodology, especially in contrast with the use of the stepwise refinement technique.

In Sect. 2 and 3 we derive, using transformations, a program for computing the Fibonacci function in logarithmic time, and we compare it with other algorithms for achieving the same efficiency. In Sect. 4 and 5 we extend this result to the general case of linear recurrence relations, and we show some interesting features of the program transformation technique. It turns out that the rules and the strategies one uses in transforming programs are powerful enough to lead in the general case also, to an efficient algorithm in a quite straightforward way. No *a priori* knowledge of matrix theory is necessary as it would have been the case for writing a fast algorithm for evaluating linear recurrence relations using the stepwise refinement technique [34]. We use indeed a mild generalization of the transformation technique à la Burstall-Darlington, because in introducing new functions we allow i) *definitions by cases* and ii) *implicit definitions*, somewhat similar to those in [2] for Boyer-Moore proof methods.

2. Deriving an Algorithm for Computing the Fibonacci Function in Logarithmic Time

We start off from the familiar definition:

Program P1

$$\begin{cases} 1.\ \mathrm{fib}(0)=1 \\ 2.\ \mathrm{fib}(1)=1 \\ 3.\ \mathrm{fib}(n+2)=\mathrm{fib}(n+1)+\mathrm{fib}(n) \quad \text{for } n\geqq 0 \end{cases}$$

and we look for a logarithmic running time program for computing the Fibonacci function. (In [8] a linear running time program is obtained). We will use the transformation method and its strategies for deriving "new and more efficient" programs from "old" ones.

As it has been the case for many other functions (see for example the factorial function in [9]), in order to improve the running time efficiency we can apply the "generalization strategy" [9], by transforming the *constant* 1 in Eq. 1 and 2 into two *variables*, say a_0 and a_1.

These variables will be considered as extra arguments of a new function as usually done in using the generalization strategy [9]. Therefore we get the following generalized Fibonacci function:

4. $G(a_0, a_1, 0) = a_0$ (GENERALIZATION EUREKA)
5. $G(a_0, a_1, 1) = a_1$
6. $G(a_0, a_1, n+2) = G(a_0, a_1, n+1) + G(a_0, a_1, n)$ for $n \geqq 0$.

The word EUREKA which annotates the above generalization step is used throughout this paper as in [8], for denoting "unobvious steps in the transformations" of our programs.

Equations 4, 5 and 6 define the new function *G by cases.*

The function G allows to compute the Fibonacci function starting from any two initial values a_0 and a_1.

Obviously we can derive the following program for computing fib(n):

7. $\text{fib}(n) = G(1, 1, n)$ for $n \geqq 0$ *Program P1.1*
4. $G(a_0, a_1, 0) = a_0$
5. $G(a_0, a_1, 1) = a_1$
6. $G(a_0, a_1, n+2) = G(a_0, a_1, n+1) + G(a_0, a_1, n)$ for $n \geqq 0$

Looking for a fast way of computing G, we need to relate $G(a_0, a_1, n+2)$ not to $G(a_0, a_1, n+1)$ and $G(a_0, a_1, n)$ as in Eq. 6, but to "more distant" values, say $G(a_0, a_1, n)$ and $G(a_0, a_1, n-1)$. This can be achieved by using the "unfolding rule" [9]. We get:

$$\begin{aligned} G(a_0, a_1, n+2) &= G(a_0, a_1, n+1) + G(a_0, a_1, n) \\ &= 2 \cdot G(a_0, a_1, n) + G(a_0, a_1, n-1) \\ &\quad \text{by unfolding } G(a_0, a_1, n+1) \text{ using 6.} \end{aligned}$$

We can iterate this unfolding process and we get:

$$G(a_0, a_1, n+2) = 3 \cdot G(a_0, a_1, n-1) + 2 \cdot G(a_0, a_1, n-2) \text{ by unfolding } G(a_0, a_1, n)$$

in the previous expression. Eventually we get:

$$G(a_0, a_1, n+2) = c_1 \cdot G(a_0, a_1, 1) + c_2 \cdot G(a_0, a_1, 0) = c_1 \cdot a_1 + c_2 \cdot a_0,$$

where the two values c_1 and c_2 depend on n, i.e. the "distance" of $G(a_0, a_1, n+2)$ from $G(a_0, a_1, 1)$ and $G(a_0, a_1, 0)$. This reasoning motivates the following "eureka step", by which we generalize the constants c_1 and c_2 and introduce an *implicit* definition of two new functions $p(n)$ and $q(n)$:

8. $G(a_0, a_1, n) = p(n) \cdot a_0 + q(n) \cdot a_1$ (LINEAR COMBINATION EUREKA)

The program transformation process continues, as usual, by looking for an explicit definition of the newly introduced functions $p(n)$ and $q(n)$.

From 4 and 8 (for $n=0$) we get: $a_0 = p(0) \cdot a_0 + q(0) \cdot a_1$.

Since this equality should hold for all values of a_0 and a_1, we have $q(0)=0$ and hence $p(0)=1$.

Analogously from 5 and 8 (for $n=1$) we get: $a_1 = p(1) \cdot a_0 + q(1) \cdot a_1$, which yields $p(1)=0$ and $q(1)=1$.

From 6 and 8 (for $n+2$ instead of n) we get:

$$G(a_0, a_1, n+1)+G(a_0, a_1, n)=p(n+2)\cdot a_0+q(n+2)\cdot a_1.$$

By 8 we have:
$p(n+1)\cdot a_0+q(n+1)\cdot a_1+p(n)\cdot a_0+q(n)\cdot a_1=p(n+2)\cdot a_0+q(n+2)$. a_1 and therefore, since this equality should hold for all values of a_0 and a_1, we get: $p(n+2)=p(n+1)+p(n)$ and $q(n+2)=q(n+1)+q(n)$.

By 4, 5 and 6 we get: $p(n)=G(1,0,n)$ and $q(n)=G(0,1,n)$. Thus the LINEAR COMBINATION EUREKA can be rewritten as follows:

8′. $G(a_0, a_1, n)=G(1,0,n)\cdot a_0+G(0,1,n)\cdot a_1$.

Since the function $G(a_0, a_1, n)$ computes the Fibonacci function starting from the initial values a_0 and a_1, if we substitute in 8′ $\mathrm{fib}(n)$ for a_0 and $\mathrm{fib}(n+1)$ for a_1 we get that $G(a_0, a_1, n)$ is equal to $\mathrm{fib}(2n)$; thus:

8″. $\mathrm{fib}(2n)=G(\mathrm{fib}(n), \mathrm{fib}(n+1), n)$
$=G(1,0,n)\cdot \mathrm{fib}(n)+G(0,1,n)\cdot \mathrm{fib}(n+1)$ by 8′.

This equation allows us to derive a logarithmic running time algorithm for computing the Fibonacci function, because the argument $2n$ has been divided by 2 and because analogous equations can be derived for $G(1,0,n)$ and $G(0,1,n)$.

However, in order to obtain Eq. 8″ using the program transformation technique, we need a second generalization step, by which we will obtain a general formula for computing $G(a_0, a_1, n)$ in logarithmic time for any value of the variables a_0, a_1 and n.

From Eq. 6 we can generalize the *constant* 2 occurring in $G(a_0, a_1, n+2)$ into a new *variable*, say k, which will be, as usual, an extra argument of the resulting function. Therefore we define the function $F(a_0, a_1, n, k)$ as follows:

9. $F(a_0, a_1, n, k)=G(a_0, a_1, n+k)$ (GENERALIZATION EUREKA defining F).

Now we look for the explicit definition of F, in terms of the argument k just introduced. We obtain:

$$\begin{aligned}
F(a_0, a_1, n, 0) &= G(a_0, a_1, n)\\
F(a_0, a_1, n, 1) &= G(a_0, a_1, n+1)\\
F(a_0, a_1, n, k+2) &= G(a_0, a_1, n+k+2)\\
&= G(a_0, a_1, n+k+1)+G(a_0, a_1, n+k) && \text{by unfolding (using 6)}\\
&= F(a_0, a_1, n, k+1)+F(a_0, a_1, n, k) && \text{by folding (using 9).}
\end{aligned}$$

Now it is possible to make for F the same transformation steps made for G, because F and G obey the same recurrence relation.

We can apply for F the "linear combination eureka" (see Eq. 8), which is now $F(a_0, a_1, n, k)=r(k)\cdot G(a_0, a_1, n)+s(k)\cdot G(a_0, a_1, n+1)$, because $G(a_0, a_1, n)$, $G(a_0, a_1, n+1)$ and k play for F the role of a_0, a_1 and n respectively for G.

Looking for the explicit definition of $r(k)$ and $s(k)$ we obtain (by making the same derivations performed for the function G) that:

$$r(0)=1, \quad r(1)=0, \quad r(k+2)=r(k+1)+r(k)$$

and

$$s(0)=0, \quad s(1)=1, \quad s(k+2)=s(k+1)+s(k).$$

Thus we have as before:

$$r(k)=G(1,0,k) \quad \text{and} \quad s(k)=G(0,1,k).$$

We get:

$$F(a_0,a_1,n,k)=G(1,0,k)\cdot G(a_0,a_1,n)+G(0,1,k)\cdot G(a_0,a_1,n+1)$$

and from it we obtain, using 9, the following equation:

9′. $G(a_0,a_1,n+k)=G(1,0,k)\cdot G(a_0,a_1,n)+G(0,1,k)\cdot G(a_0,a_1,n+1)$

(EFFICIENCY EQUATION)

Equation 9′ allows us to achieve the logarithmic running time algorithm we were looking for. In fact, if we want to compute $\text{fib}(2k)$ we need to compute $G(1,1,2k)$ (by Eq. 7) and for computing $G(1,1,2k)$ according to Eq. 9′, we need only to know $G(1,0,k)$, $G(1,1,k)$, $G(0,1,k)$ and $G(1,1,k+1)$ which are values of the function G around the point k. As we already mentioned, this division by 2 of the relevant argument allows the desired efficiency.

From Eq. 9′ we can derive the new program P2 for computing the Fibonacci function:

	Program P2
10. $\text{fib}(0)=1$	by 1
11. $\text{fib}(1)=1$	by 2
12. $\text{fib}(2k)=G(1,1,2k)$	by 7
$=G(1,0,k)\cdot G(1,1,k)+G(0,1,k)\cdot G(1,1,k+1)$	by 9′
13. $\text{fib}(2k+1)=G(1,1,2k+1)$	by 7
$=G(1,0,k)\cdot G(1,1,k+1)+G(0,1,k)\cdot G(1,1,k+2)$	by 9′
14. $G(a_0,a_1,0)=a_0$	by 4
15. $G(a_0,a_1,1)=a_1$	by 5
16. $G(a_0,a_1,2k)=G(1,0,k)\cdot G(a_0,a_1,k)$	by 9′
$+G(0,1,k)\cdot G(a_0,a_1,k+1)$	
17. $G(a_0,a_1,2k+1)=G(1,0,k)\cdot G(a_0,a_1,k+1)$	by 9′
$+G(0,1,k)\cdot G(a_0,a_1,k+2)$	

Unfortunately, as one can see from Fig. 1, program P2 may evoke computations with redundant evaluations. And in order to guarantee a logarithmic running time for P2 one should avoid them (or, at least, prove them to be suitably bounded).

Notice that redundant evaluations can be detected by symbolic evaluation alone, using "unfolding" [8]: in our case, for example, when computing $\text{fib}(2k)$, we will compute $G(1,0,k/2)$ 4 times (see Fig. 1).

But this difficulty can be easily solved by using the "tupling strategy" [26] also called "pairing strategy" in [9]. The "tupling strategy" consists in defining a new function as the tuple of all functions which require common subcomputa-

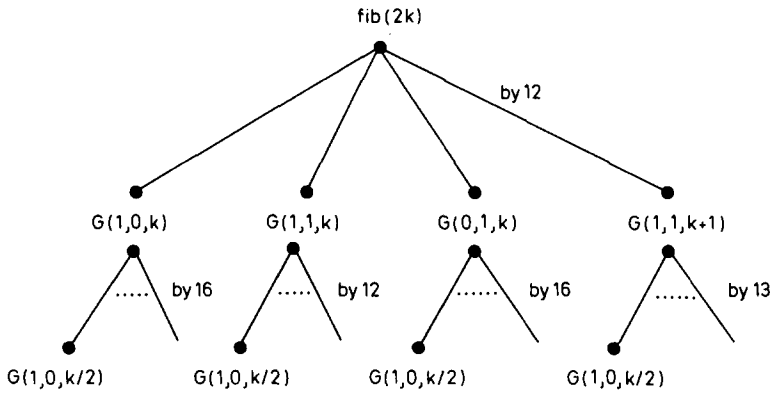


Fig. 1. Redundant computations in program P2: $G(1,0,k/2)$ occurs 4 times in evaluating fib$(2k)$. (Suppose k even.)

tions. We will apply it by "tupling" together $G(1,0,k)$, $G(1,1,k)$, $G(0,1,k)$ and $G(1,1,k+1)$ because they all need the computation of $G(1,0,k/2)$ (see Fig. 1).

Therefore we will define the following auxiliary function $t(k)$:

18. $t(k)=\langle G(1,0,k),\ G(0,1,k),\ G(1,1,k),\ G(1,1,k+1)\rangle$ (TUPLING EUREKA)

The order in which the individual functions G's occur in $t(k)$ is immaterial. The explicit definition of the function $t(k)$ can easily be derived by standard applications of the "folding" and "unfolding" rules using Eqs. 4, 5, 6 and 9′ and we get:

19. $t(0)=\langle 1,0,1,1\rangle$

20. $t(2k)=\langle G(1,0,2k),\ G(0,1,2k),\ G(1,1,2k),\ G(1,1,2k+1)\rangle$
$=\langle G(1,0,k)\cdot G(1,0,k)+G(0,1,k)\cdot G(1,0,k+1),$
$G(1,0,k)\cdot G(0,1,k)+G(0,1,k)\cdot G(0,1,k+1),$
$G(1,0,k)\cdot G(1,1,k)+G(0,1,k)\cdot G(1,1,k+1),$
$G(1,0,k)\cdot G(1,1,k+1)+G(0,1,k)\cdot G(1,1,k+2)\rangle$ *by* 9′
$=\langle a^2+b^2,\ b(2a+b),\ ac+bd,\ ad+b(c+d)\rangle$
where $\langle a,b,c,d\rangle=t(k)$ *by* 6 *and* 9′

21. $t(2k+1)=\langle G(1,0,2k+1),\ G(0,1,2k+1),\ G(1,1,2k+1),\ G(1,1,2k+2)\rangle$
$=\langle 2ab+b^2,\ (a+b)^2+b^2,\ ad+b(c+d),\ a(c+d)+b(2d+c)\rangle$
where $\langle a,b,c,d\rangle=t(k)$

Therefore we can now obtain the following nonredundant program P3, which computes the Fibonacci function in logarithmic time.

Program P3

10. fib$(0)=1$
11. fib$(1)=1$
12′. fib$(2k)=ac+bd$ **where** $\langle a,b,c,d\rangle=t(k)$
13′. fib$(2k+1)=ad+b(c+d)$ **where** $\langle a,b,c,d\rangle=t(k)$
19. $t(0)=\langle 1,0,1,1\rangle$
20. $t(2k)=\langle a^2+b^2,\ b(2a+b),\ ac+bd,\ ad+b(c+d)\rangle$
where $\langle a,b,c,d\rangle=t(k)$
21. $t(2k+1)=\langle 2ab+b^2,\ (a+b)^2+b^2,\ ad+b(c+d),\ a(c+d)+b(2d+c)\rangle$
where $\langle a,b,c,d\rangle=t(k)$

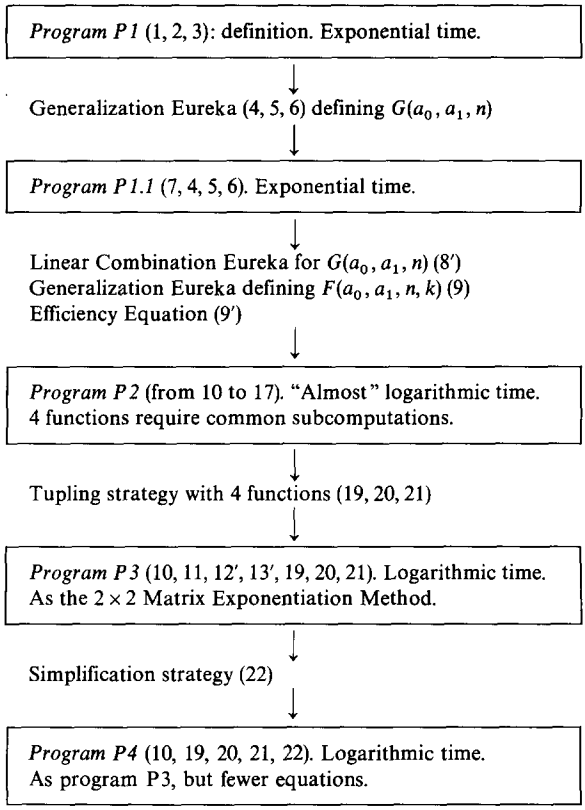


Fig. 2. Derivation of logarithmic running time algorithms for computing the Fibonacci function. (We have annotated the transformation steps and the programs with the corresponding equation numbers.)

Notice that, instead of "tupling" the functions which cause redundant computations in Eq. 12, namely $G(1, 0, k)$,$G(0, 1, k)$, $G(1, 1, k)$ and $G(1, 1, k+1)$, as we did, we could have tupled the functions $G(1, 0, k)$, $G(0, 1, k)$, $G(1, 1, k+1)$ and $G(1, 1, k+2)$ which cause redundant computations in Eq. 13. The resulting program would have been equivalent to program P3.

We also can simplify program P3 using the following equation which can be derived from Eqns. 7 and 18:

22. $\text{fib}(n+1) = \pi 3(t(n))$

where, in general, $\pi i(\langle e0, e1, \ldots, ek\rangle) = ei$ for $i = 0, \ldots, k$.
(Notice that πi denotes the $(i+1)$st projection function. It is not a usual convention, but it will shorten our notations in what follows.)

We obtain program P4.

Program P4

10. $\text{fib}(0) = 1$
22. $\text{fib}(n+1) = \pi 3(t(n))$ **where** $\pi 3(\langle e0, e1, e2, e3\rangle) = e3$
19. $t(0) = \langle 1, 0, 1, 1\rangle$

20. $t(2k)=\langle a^2+b^2,\ b(2a+b),\ ac+bd,\ ad+b(c+d)\rangle$
where $\langle a,b,c,d\rangle=t(k)$

21. $t(2k+1)=\langle 2ab+b^2,\ (a+b)^2+b^2,\ ad+b(c+d),\ a(c+d)+b(2d+c)\rangle$
where $\langle a,b,c,d\rangle=t(k)$

We could have also derived the equation:

22′. $\text{fib}(n)=\pi 2(t(n))$ for $n\geqq 0$

and we could have used it in program P4 instead of Eqs. 22 and 10. But, in evaluating $\pi 2(t(n))$ we should avoid computing all 4 components of $t(n)$, because otherwise we would also compute $\pi 3(t(n))$ which is equal to $\text{fib}(n+1)$, and it seems awkward to compute $\text{fib}(n+1)$ while computing $\text{fib}(n)$. A solution to this inconvenience can be obtained by using a "call by need" evaluation mode for the projection function $\pi 2$.

In Fig. 2 the reader may have a synoptic account of the program transformation steps we have made so far.

3. A Comparison with Other Algorithms

Linear recurrence relations can also be evaluated in logarithmic time using the well-known *matrix exponentiation method* [19]. We will recall it in the case of the Fibonacci function evaluation. Directly from the definition, we have:

$$\begin{bmatrix}\text{fib}(2)\\ \text{fib}(1)\end{bmatrix}=\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\cdot\begin{bmatrix}\text{fib}(1)\\ \text{fib}(0)\end{bmatrix}$$

and, in general, we have:

$$\begin{bmatrix}\text{fib}(k+1)\\ \text{fib}(k)\end{bmatrix}=M^k\cdot\begin{bmatrix}\text{fib}(1)\\ \text{fib}(0)\end{bmatrix}=M^k\cdot\begin{bmatrix}1\\ 1\end{bmatrix}\quad\text{for } k\geqq 0\quad\text{where } M=\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}.$$

Therefore, in order to compute $\text{fib}(k+1)$ we have to compute M^k. This matrix can be computed in logarithmic time, because we can compute the values of $M, M^2, M^4, M^8, \ldots$ by successively squaring M and then multiply those matrices whose exponents contribute to a sum equal to k [19, 29]. More formally, we have: $M^k=\prod_{i=0}^{n} M^{b_i 2^i}$, where the b_i's are defined by the binary expansion of k, i.e. $k=\sum_{i=0}^{n} b_i 2^i$.

Example 1. For computing $\text{fib}(14)$, since $13=1+4+8$, we have:

$$\begin{bmatrix}\text{fib}(14)\\ \text{fib}(13)\end{bmatrix}=M^{13}\cdot\begin{bmatrix}1\\ 1\end{bmatrix}=M\cdot M^4\cdot M^8\cdot\begin{bmatrix}1\\ 1\end{bmatrix}=\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\cdot\begin{bmatrix}5 & 3\\ 3 & 2\end{bmatrix}\cdot\begin{bmatrix}34 & 21\\ 21 & 13\end{bmatrix}\cdot\begin{bmatrix}1\\ 1\end{bmatrix}$$
$$=\begin{bmatrix}377 & 233\\ 233 & 144\end{bmatrix}\cdot\begin{bmatrix}1\\ 1\end{bmatrix}$$

Thus $\text{fib}(14)=377+233=610$. □

What kind of relationship exists between the matrix exponentiation method and our method as defined by program P4? The answer to this question is given by the following fact, which shows that the 4-tuple $t(k)$ directly corresponds to M^k for any $k \geqq 0$.

Fact 1. *Let* M *be* $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$, M^k *be* $\begin{bmatrix} c & b \\ d & a \end{bmatrix}$ *and* $t(k)$ *be defined as in program P4.* $\forall k \geqq 0\ t(k) = \langle a, b, c, c+d \rangle$.

Proof. By induction on k. □

It is very interesting to notice that the use of the tupling strategy, which we applied in the transformation approach only for avoiding redundant evaluations of common subexpressions, "rediscovered", as Fact 1 indicates, the matrix exponentiation approach, by satisfying the apparently unrelated requirement of improving efficiency. But the relationship between the two approaches can be shown to be even stronger, as we will now indicate.

The matrix exponentiation method can be improved because the following Fact 2 allows us to derive the matrix M^{2k} from the matrix M^k, for any $k \geqq 0$, by computing only 2 elements of it.

Fact 2. *Under the hypotheses of Fact 1,* $\forall k \geqq 0$ $d = b$ *and* $c = a + b$, *i.e. the matrices* M^k*s are symmetric and they have only two independent elements, namely a and b.*

Proof. By induction on k. □

Can we "rediscover" the improved version of the matrix exponentiation method (in which we compute only two elements of the matrices M^k, for any $k \geqq 0$, by using Fact 2) by applying the tupling strategy for avoiding redundant computations so that only two functions (instead of 4 as we did in the definition of $t(k)$) are paired together? The answer is "yes" and the corresponding program can be obtained as follows from program P2. The technique used is an application of what we may call a *simplification strategy*. We can transform equations 12 and 13 for minimizing the number of distinct values occurring in them.

$$
\begin{aligned}
12''.\ \mathrm{fib}(2k) &= G(1,0,k) \cdot G(1,1,k) + G(0,1,k) \cdot G(1,1,k+1) && \text{by 12} \\
&= G(1,0,k) \cdot (G(1,0,k) \cdot G(1,1,0) + G(0,1,k) \cdot G(1,1,1)) \\
&\quad + G(0,1,k) \cdot (G(1,0,k) \cdot G(1,1,1) + G(0,1,k) \cdot G(1,1,2)) && \text{by 9'} \\
&= (a+b)^2 + b^2 \ \textbf{where}\ a, b = G(1,0,k), G(0,1,k) && \text{by 4, 5 and 6.}
\end{aligned}
$$

Analogously:

$$
\begin{aligned}
13''.\ \mathrm{fib}(2k+1) &= G(1,0,k) \cdot G(1,1,k+1) + G(0,1,k) \cdot G(1,1,k+2) && \text{by 13} \\
&= (a+b)^2 + 2b(a+b) \ \textbf{where}\ a, b = G(1,0,k), G(0,1,k) && \text{by 9'.}
\end{aligned}
$$

Analogously we may transform equations 16 and 17 and we get:

$$
\begin{aligned}
G(a_0, a_1, k) &= G(1,0,k) \cdot a_0 + G(0,1,k) \cdot a_1 && \text{by 8'} \\
G(a_0, a_1, k+1) &= G(1,0,k) \cdot G(a_0, a_1, 1) + G(0,1,k) \cdot G(a_0, a_1, 2) && \text{by 9'} \\
&= G(1,0,k) \cdot a_1 + G(0,1,k) \cdot (a_0 + a_1) && \text{by 4, 5 and 6}
\end{aligned}
$$

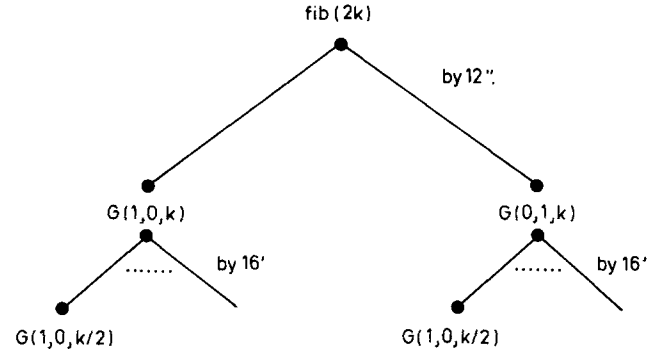


Fig. 3. Redundant computations in program P2.1: $G(1, 0, k/2)$ occurs twice in evaluating fib$(2k)$. (Suppose k even.)

$$
\begin{aligned}
G(a_0, a_1, k+2) &= G(a_0, a_1, k+1) + G(a_0, a_1, k) && \text{by 6}\\
&= G(1,0,k)\cdot a_1 + G(0,1,k)\,(a_0+a_1) + G(a_0, a_1, k) && \text{by above equation for } G(a_0, a_1, k+1)\\
&= G(1,0,k)\,a_1 + G(0,1,k)\,(a_0+a_1)\\
&\quad + G(1,0,k)\,G(a_0, a_1, 0) + G(0,1,k)\,G(a_0, a_1, 1) && \text{by 9'}\\
&= G(1,0,k)\,(a_0+a_1) + G(0,1,k)\,(a_0+2\,a_1) && \text{by 4 and 5.}
\end{aligned}
$$

Therefore we get the following program P2.1 from program P2:

Program P2.1

10. $\text{fib}(0) = 1$
11. $\text{fib}(1) = 1$
12''. $\text{fib}(2k) = (a+b)^2 + b^2$ **where** $a, b = G(1,0,k), G(0,1,k)$
13''. $\text{fib}(2k+1) = (a+b)^2 + 2b(a+b)$ **where** $a, b = G(1,0,k), G(0,1,k)$
14. $G(a_0, a_1, 0) = a_0$
15. $G(a_0, a_1, 1) = a_1$
16'. $G(a_0, a_1, 2k) = G(1,0,k)\cdot G(a_0, a_1, k) + G(0,1,k)\cdot G(a_0, a_1, k+1)$ by 9'
$= p^2\,a_0 + 2\,a_1\,pq + (a_0+a_1)\,q^2$
where $p, q = G(1,0,k), G(0,1,k)$
17'. $G(a_0, a_1, 2k+1) = G(1,0,k)\cdot G(a_0, a_1, k+1) + G(0,1,k)\cdot G(a_0, a_1, k+2)$ by 9'
$= p^2\,a_1 + 2(a_0+a_1)\,pq + (a_0+2\,a_1)\,q^2$
where $p, q = G(1,0,k), G(0,1,k)$

Program P2.1 still suffers from the presence of redundant computations (see Fig. 3).

They are caused by functions $G(1, 0, k)$ and $G(0, 1, k)$ and therefore we will use the "tupling strategy" defining the following auxiliary function $r(k)$:

23. $r(k) = \langle G(1,0,k), G(0,1,k)\rangle$ (TUPLING EUREKA)
24. $r(0) = \langle 1, 0\rangle$
25. $r(2\,k) = \langle G(1,0,2\,k), G(0,1,2\,k)\rangle$
$= \langle p^2+q^2, 2\,pq+q^2\rangle$ **where** $\langle p, q\rangle = r(k)$ by 16'

26. $r(2k+1)=\langle G(1,0,2k+1), G(0,1,2k+1)\rangle$
$=\langle 2pq+q^2, (p+q)^2+q^2\rangle$ **where** $\langle p,q\rangle=r(k)$ by 17′

Therefore we can transform program P2.1 into program P3.1 where no redundant computations occur and, as required, we tupled only two functions. The running time for P3.1 is again logarithmic.

Program P3.1

10. $\text{fib}(0)=1$
11. $\text{fib}(1)=1$
12″. $\text{fib}(2k)=(a+b)^2+b^2$ **where** $\langle a,b\rangle=r(k)$
13″. $\text{fib}(2k+1)=(a+b)^2+2b(a+b)$ **where** $\langle a,b\rangle=r(k)$
24. $r(0)=\langle 1,0\rangle$
25. $r(2k)=\langle a^2+b^2, 2ab+b^2\rangle$ **where** $\langle a,b\rangle=r(k)$
26. $r(2k+1)=\langle 2ab+b^2, (a+b)^2+b^2\rangle$ **where** $\langle a,b\rangle=r(k)$

The direct correspondence between program P3.1 and the matrix exponentiation method is stated in the following fact:

Fact 3. *Let* M *be* $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$, M^k *be* $\begin{bmatrix}c & b\\ d & a\end{bmatrix}$ *and* $r(k)$ *be defined as in program P3.1.* $\forall k\geqq 0\ r(k)=\langle a,b\rangle$.

Proof. By induction on k. □

As we simplified program P3 and obtained program P4 we can simplify program P3.1 also, by using once more the efficiency equation 9′.

7′. $\text{fib}(n)=G(1,1,n)$ for $n\geqq 0$ by 7
$=G(1,0,n)\cdot G(1,1,0)+G(0,1,n)\cdot G(1,1,1)$ for $n\geqq 0$ by 9′
$=G(1,0,n)+G(0,1,n)$ for $n\geqq 0$ by 4, 5
$=a+b$ *where* $\langle a,b\rangle=r(n)$ for $n\geqq 0$ by 23

Therefore we can get the following program P4.1:

Program P4.1

10. $\text{fib}(0)=1$
11. $\text{fib}(1)=1$
7′. $\text{fib}(n+2)=a+b$ **where** $\langle a,b\rangle=r(n+2)$ for $n\geqq 0$
24. $r(0)=\langle 1,0\rangle$
25. $r(2k)=\langle a^2+b^2, 2ab+b^2\rangle$ **where** $\langle a,b\rangle=r(k)$
26. $r(2k+1)=\langle 2ab+b^2, (a+b)^2+b^2\rangle$ **where** $\langle a,b\rangle=r(k)$

Notice that program P4.1 is shorter than program P3.1 but it determines exactly the same sequence of computations, as it can be proved by induction on k.

Fig. 4 below summarizes the last transformation steps we have performed.

Instead of using the simplification strategy and the tupling strategy, we can get program P3.1 directly from program P1 by discovering an efficiency equation a bit more "clever" than 9′ as follows. We could have thought of expressing $G(1,0,k)$ and $G(0,1,k)$ in 9′ in terms of the Fibonacci function for reducing the

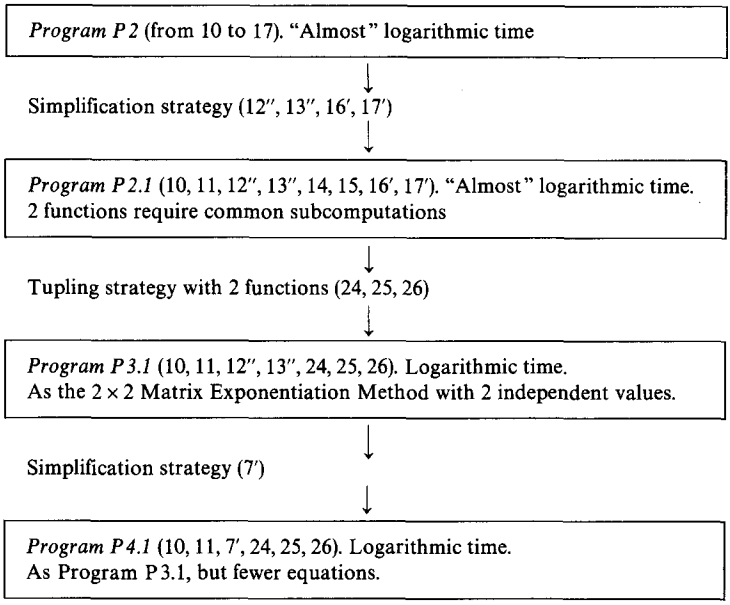


Fig. 4. Alternative derivation of logarithmic running time algorithms for the Fibonacci function. (Transformation steps and programs are annotated with the corresponding equation numbers.)

number of different functions to be computed. This can be done by extending the definition of the Fibonacci function so that $\text{fib}(n+2)=\text{fib}(n+1)+\text{fib}(n)$ holds for $n \geqq -2$. Therefore, if we want to have again $\text{fib}(0)=1$ and $\text{fib}(1)=1$, we must have $\text{fib}(-1)=0$ and $\text{fib}(-2)=1$. Thus:

27. $G(1, 0, k)=\text{fib}(k-2)$ by 4, 5 and 6
28. $G(0, 1, k)=\text{fib}(k-1)$ by 4, 5 and 6

We can then rewrite the EFFICIENCY EQUATION as follows:

9''. $G(a_0, a_1, n+k)=\text{fib}(k-2)\cdot G(a_0, a_1, n)+\text{fib}(k-1)\cdot G(a_0, a_1, n+1)$ for $n \geqq 0, k \geqq 0$

Using 9'' we get the following program for computing the Fibonacci function:

10. $\text{fib}(0) = 1$
11. $\text{fib}(1) = 1$
29. $\text{fib}(2k) = G(1, 1, 2k)$ by 7
$= \text{fib}(k-2)\cdot\text{fib}(k)+\text{fib}(k-1)\cdot\text{fib}(k+1)$ by 9'' and 7
$= (a+b)^2+b^2$ **where** $a, b=\text{fib}(k-2), \text{fib}(k-1)$ by 3
30. $\text{fib}(2k+1)=G(1, 1, 2k+1)$ by 7
$= \text{fib}(k-2)\cdot\text{fib}(k+1)+\text{fib}(k-1)\cdot\text{fib}(k+2)$ by 9'' and 7
$= (a+b)^2+2b(a+b)$ **where** $a, b=\text{fib}(k-2), \text{fib}(k-1)$ by 3

By Eqs. 23, 27 and 28 it turns out that Eqs. 29 and 30 are the same as 12'' and 13''. Therefore applying the tupling strategy by defining $r(n)=\langle\text{fib}(n-2), \text{fib}(n-1)\rangle$

we can get again program P3.1. It is very interesting to notice that the cleverness of discovering 9″, starting from 9′ (including also the extension of the Fibonacci function definition) can be obtained via the application of a very simple strategy, i.e. the simplification one. Therefore, in some particular cases, the basic strategies are indeed very powerful and allow us to obtain results which can be otherwise derived only by "intelligent" reasoning.

Linear recurrence relations may also be solved using the *generating functions method* [22]. Applying that method it is possible to derive an explicit formula for the nth element of the series which satisfies a given recurrence relation. That formula contains in general an exponentiation to the nth power which allows a logarithmic running time algorithm. For example, for the Fibonacci relation, fib(n) is equal to $(A^n - B^n)/\sqrt{5}$ where $A=(1+\sqrt{5})/2$ and $B=(1-\sqrt{5})/2$. Therefore as far as running time is concerned, the program we obtained using the transformation technique is not less efficient, at least asymptotically.

4. The General Case of Homogeneous Linear Recurrence Relations

In this section we deal with the general case of homogeneous linear recurrence relations with constant coefficients (as defined later) and we show that analogous steps to those presented for deriving an efficient algorithm for computing the Fibonacci function can be performed.

We will follow for the general case the same transformation steps indicated in Fig. 2 from program P1 to program P2 and the ones indicated in Fig. 4 from program P2 to program P3.1: in fact the *generalization eurekas* and the *linear combination eurekas* can be applied in the same manner, so that we can derive an algorithm for evaluating any homogeneous linear recurrence relation in logarithmic time. Therefore, as the matrix exponentiation method is a standard method for evaluating linear recurrence relations, so is our transformation method. This is an interesting fact because it shows the "uniform" power of transformation strategies with respect to a given class of problems, just as "paradigms" [17] in the stepwise refinement technique have a uniform power for solving problems in a given class.

Let us consider a general homogeneous linear recurrence relation of order r:

4.1.
$$\begin{cases} h(0)=h_0 \\ h(1)=h_1 \\ \quad\vdots \\ h(r-1)=h_{r-1} \\ h(n)=b_0\,h(n-r)+\ldots+b_{r-1}\,h(n-1) \quad \text{for } n\geqq r. \end{cases}$$

We will also write $h(n)=L(h(n-r),\ldots,h(n-1))$ where $L(x_0,\ldots,x_{r-1})=\sum_{i=0}^{r-1} b_i\,x_i$ is a linear polynomial in the variables $x_0,\ldots,x_{r-1}$ with the constant coefficients $b_0,\ldots,b_{r-1}$. The *generalization eureka* introduces the following function H (analogous to the function $G(a_0,a_1,n)$):

4.2. $$\begin{cases} H(h_0, \ldots, h_{r-1}, 0) = h_0 \\ \quad\vdots \\ H(h_0, \ldots, h_{r-1}, r-1) = h_{r-1} \\ H(h_0, \ldots, h_{r-1}, n) = L(H(h_0, \ldots, h_{r-1}, n-r), \ldots, H(h_0, \ldots, h_{r-1}, n-1)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } n \geqq r. \end{cases}$$

The general form of the *linear combination eureka* (analogous to Eq. 8') is:

4.3. $$H(h_0, \ldots, h_{r-1}, n) = H(1, 0, \ldots, 0, n) \cdot h_0 + \ldots + H(0, 0, \ldots, 1, n) \cdot h_{r-1} = \sum_{i=0}^{r-1} H_i(n) h_i$$

where we denoted by $H_i(n)$ the term $H(\underbrace{0, \ldots, 0}_{i}, 1, \underbrace{0, \ldots, 0}_{r-1-i}, n)$ for $i = 0, \ldots, r-1$.

If we take $h_0, \ldots, h_{r-1}$ to be equal to the r consecutive values of $H(h_0, \ldots, h_{r-1}, n)$ for $n = k, k+1, \ldots, k+r-1$, then from the linear combination eureka, we can derive the following general form of the *efficiency equation*, as we did in deriving equation 9':

4.4. $$\begin{aligned} H(h_0, \ldots, h_{r-1}, m+k) &= H(1, 0, \ldots, 0, m) \cdot H(h_0, \ldots, h_{r-1}, k) + \ldots \\ &\quad + H(0, 0, \ldots, 1, m) \cdot H(h_0, \ldots, h_{r-1}, k+r-1) \\ &= \sum_{i=0}^{r-1} H_i(m)\, H(h_0, \ldots, h_{r-1}, k+i) \end{aligned}$$

which can be proved by induction on k using Eq. 4.3.
Therefore we have:

4.4.1 $$h(2k) = H(h_0, \ldots, h_{r-1}, k+k) = \sum_{i=0}^{r-1} H_i(k)\, H(h_0, \ldots, h_{r-1}, k+i)$$

and

4.4.2 $$h(2k+1) = H(h_0, \ldots, h_{r-1}, k+k+1) = \sum_{i=0}^{r-1} H_i(k)\, H(h_0, \ldots, h_{r-1}, k+i+1)$$

which are analogous to the Eqs. 12 and 13.

For avoiding redundant computations in computing $h(2k)$ and $h(2k+1)$ we need to apply the tupling strategy (as we did in deriving equation 18) to the following $2r$ functions: $H_0(k), \ldots, H_{r-1}(k)$, $H(h_0, \ldots, h_{r-1}, k), \ldots, H(h_0, \ldots, h_{r-1}, k+r-1)$. Out of these $2r$ values, only r are independent. In fact, as it happened for the Eqs. 16' and 17', using the Eq. 4.4, we can express $H(h_0, \ldots, h_{r-1}, k+i)$ for $i = 0, \ldots, r$ in terms of the $H_j(k)$'s for $0 \leqq j \leqq r-1$. We have:

4.5 $$H(h_0, \ldots, h_{r-1}, k+i) = \sum_{j=0}^{r-1} H_j(k)\, H(h_0, \ldots, h_{r-1}, i+j) \quad \text{for } 0 \leqq i \leqq r \quad \text{by 4.4.}$$

Therefore we can rewrite the Eqs. 4.4, 4.4.1 and 4.4.2 as follows:

4.4'. $$H(h_0, \ldots, h_{r-1}, m+k) = \sum_{i=0}^{r-1} H_i(m) \left[\sum_{j=0}^{r-1} H_j(k)\, H(h_0, \ldots, h_{r-1}, i+j) \right],$$

4.4.1′. $h(2k)=\sum_{i=0}^{r-1} H_i(k) \sum_{j=0}^{r-1} H_j(k)\, H(h_0, \ldots, h_{r-1}, i+j)=\sum_{i=0}^{r-1} H_i(k) \left[\sum_{j=0}^{r-1} H_j(k) h_{i+j}\right]$,

4.4.2′. $h(2k+1)=\sum_{i=0}^{r-1} H_i(k) \sum_{j=0}^{r-1} H_j(k)\, H(h_0, \ldots, h_{r-1}, i+j+1)$

$$=\sum_{i=0}^{r-1} H_i(k) \left[\sum_{j=0}^{r-1} H_j(k)\, h_{i+j+1}\right]$$

where h_i stands for $h(i)$ for $0 \leqq i \leqq 2r-1$.

The h_i's for $0 \leqq i \leqq 2r-1$ can be computed once and for all, using the equations 4.1. Then we apply the tupling strategy, as we did in Eq. 23, for computing the r independent values by defining the following function $t(k)$:

4.6. $t(k)=\langle H(1, 0, \ldots, 0, k), H(0, 1, \ldots, 0, k), \ldots, H(0, 0, \ldots, 1, k)\rangle$

$=\langle H_0(k), H_1(k), \ldots, H_{r-1}(k)\rangle$ (TUPLING EUREKA).

Now we can derive the equations for computing $t(k)$ using Eq. 4.6 and the efficiency equation 4.4′, in much the same way as we derived equations 24, 25 and 26.

4.7. $t(0)=\langle H_0(0), H_1(0), \ldots, H_{r-1}(0)\rangle$ by 4.6

$=\langle 1, 0, \ldots, 0\rangle$ by 4.2

$\vdots$

4.8. $t(r-1)=\langle H_0(r-1), H_1(r-1), \ldots, H_{r-1}(r-1)\rangle$ by 4.6

$=\langle 0, 0, \ldots, 1\rangle$ by 4.2

4.9. $t(2k)=\langle H_0(k+k), \ldots, H_{r-1}(k+k)\rangle$ by 4.6

$$=\left\langle \sum_{i=0}^{r-1} H_i(k) \left[\sum_{j=0}^{r-1} H_j(k)\, H_0(i+j)\right], \ldots, \sum_{i=0}^{r-1} H_i(k) \left[\sum_{j=0}^{r-1} H_j(k)\, H_{r-1}(i+j)\right]\right\rangle$$

by 4.4′

4.10. $t(2k+1)=\langle H_0(k+k+1), \ldots, H_{r-1}(k+k+1)\rangle$ by 4.6

$$=\left\langle \sum_{i=0}^{r-1} H_i(k) \cdot H_0(k+1+i), \ldots, \sum_{i=0}^{r-1} H_i(k) \cdot H_{r-1}(k+1+i)\right\rangle$$

by 4.4

$$=\left\langle \sum_{i=0}^{r-1} H_i(k) \left[\sum_{j=0}^{r-1} H_j(k)\, H_0(i+j+1)\right], \ldots, \right.$$

$$\left. \sum_{i=0}^{r-1} H_i(k) \left[\sum_{j=0}^{r-1} H_j(k)\, H_{r-1}(i+j+1)\right]\right\rangle$$

by 4.5

We have now completed the task of deriving a general program (given below) for evaluating any homogeneous linear recurrence relation of order r in logarithmic time.

Recursive program for evaluating linear recurrence relations (as given by Eqs. 4.1)

$$h(0) = h_0 \qquad \text{by 4.1}$$
$$\vdots$$
$$h(r-1) = h_{r-1} \qquad \text{by 4.1}$$
$$h(2k) = \sum_{i=0}^{r-1} a_i \sum_{j=0}^{r-1} a_j\, h(i+j) \quad \textbf{where } \langle a_0, \ldots, a_{r-1}\rangle = t(k) \quad \text{by 4.4.1' and 4.6}$$
$$h(2k+1) = \sum_{i=0}^{r-1} a_i \sum_{j=0}^{r-1} a_j\, h(i+j+1) \quad \textbf{where } \langle a_0, \ldots, a_{r-1}\rangle = t(k) \quad \text{by 4.4.2' and 4.6}$$
$$t(0) = \langle 1, 0, \ldots, 0\rangle \qquad \text{by 4.7}$$
$$\vdots$$
$$t(r-1) = \langle 0, 0, \ldots, 1\rangle \qquad \text{by 4.8}$$
$$t(2k) = \left\langle \sum_{i=0}^{r-1} a_i \left[\sum_{j=0}^{r-1} a_j \cdot \pi 0(t(i+j))\right], \ldots, \sum_{i=0}^{r-1} a_i \left[\sum_{j=0}^{r-1} a_j \cdot \pi r-1(t(i+j))\right]\right\rangle$$
$$\textbf{where } \langle a_0, \ldots, a_{r-1}\rangle = t(k) \quad \text{by 4.9}$$
$$t(2k+1) = \left\langle \sum_{i=0}^{r-1} a_i \left[\sum_{j=0}^{r-1} a_j \cdot \pi 0(t(i+j+1))\right], \ldots, \sum_{i=0}^{r-1} a_i \left[\sum_{j=0}^{r-1} a_j \cdot \pi r-1(t(i+j+1))\right]\right\rangle$$
$$\textbf{where } \langle a_0, \ldots, a_{r-1}\rangle = t(k) \quad \text{by 4.10}$$

Remarks

1. The precomputation of the $h(i)$'s and the $t(i)$'s for $r \leqq i \leqq 2r-1$ is required using Eqs. 4.1 and Definitions 4.2 and 4.6.
2. $\pi i(\langle a_0, \ldots, a_{r-1}\rangle) = a_i$ for $i = 0, \ldots, r-1$.

As a consequence of the program we have just derived one can say that, in general, any homogeneous linear recurrence relation of order r (in which a generic value depends on r preceding values) can be evaluated in logarithmic time if we compute r values simultaneously (see Eq. 4.6).

Example 2. Given the following linear recurrence relation:

$$\begin{cases} p(0) = 1 \\ p(n) = 2\, p(n-1) \quad \text{for } n > 0 \end{cases}$$

we have $r = 1$ and *therefore* all summation operators *occurring* in the general program vanish.

Moreover since $r = 2r - 1 = 1$ we have to compute only $p(1)$, which is equal to 2. We have:

$$p(2k) = a_0 \cdot a_0 \cdot p(0) = a_0^2$$
$$p(2k+1) = a_0 \cdot a_0 \cdot p(1) = 2\, a_0^2$$
$$t(k) = \langle a_0\rangle;\ t(0) = \langle 1\rangle;\ t(2k) = \langle a_0^2\rangle;\ t(2k+1) = \langle 2\, a_0^2\rangle$$

and therefore we get (by forgetting the unit-tupling operator):

$$\begin{cases} p(0) = 1 \\ p(2k) = a_0^2 \quad \textbf{where } a_0 = t(k) \\ p(2k+1) = 2\, a_0^2 \quad \textbf{where } a_0 = t(k) \end{cases}$$

$$\begin{cases} t(0)=1 & \\ t(2k)=a_0^2 & \textbf{where } a_0=t(k) \\ t(2k+1)=2a_0^2 & \textbf{where } a_0=t(k). \end{cases}$$

Since $p(k)$ and $t(k)$ obey the same equations, $p(k)=t(k)$. Thus the program we obtained can be further simplified as follows:

$$\begin{cases} p(0)=1 \\ p(2k)=p^2(k) \\ p(2k+1)=2\cdot p^2(k). \end{cases} \quad \square$$

Example 3. Let us consider the following recurrence relation:

$$\begin{cases} d(0)=1 \\ d(1)=2 \\ d(2)=0 \\ d(n)=d(n-1)+2\,d(n-3) & \text{for } n>2. \end{cases}$$

We have: $r=3, 2r-1=5$. Thus we have to compute $d(i)$ for $i=3, 4, 5$ and we get $d(3)=2$, $d(4)=d(5)=6$. We also have:

$$t(0)=\langle 1,0,0\rangle,\ t(1)=\langle 0,1,0\rangle,\ t(2)=\langle 0,0,1\rangle,\ t(3)=\langle 2,0,1\rangle,$$
$$t(4)=\langle 2,2,1\rangle,\ t(5)=\langle 2,2,3\rangle$$

because for $i=3, 4, 5$ and $j=0, 1, 2$: $\pi j(t(i))=\pi j(t(i-1))+2\,\pi j(t(i-3))$.
(In fact the components of $t(i)$ satisfy the same recurrence relation given for $d(n)$, as one can see from Eqs. 4.2 and 4.6). We can then derive:

$$\begin{aligned} d(2k) &= a_0\cdot(a_0\,d(0)+a_1\,d(1)+a_2\,d(2))+a_1(a_0\,d(1)+a_1\,d(2)+a_2\,d(3)) \\ &\quad +a_2(a_0\,d(2)+a_1\,d(3)+a_2\,d(4)) \\ &= a_0^2+4a_0a_1+4a_1a_2+6a_2^2 \qquad \textbf{where } \langle a_0,a_1,a_2\rangle=t(k) \end{aligned}$$

and analogously:

$$d(2k+1)=2a_0^2+2a_1^2+6a_2^2+4a_0a_2+12a_1a_2 \qquad \textbf{where } \langle a_0,a_1,a_2\rangle=t(k).$$

We also have:

$$t(2k)=\langle a_0^2+4a_1a_2+2a_2^2,\ 2a_0a_1+2a_2^2,\ a_1^2+a_2^2+2a_0a_2+2a_1a_2\rangle$$
$$\textbf{where } \langle a_0,a_1,a_2\rangle=t(k)$$

and

$$t(2k+1)=\langle 4a_0a_2+2a_1^2+4a_1a_2+2a_2^2,\ a_0^2+4a_1a_2+2a_2^2,$$
$$2a_0a_1+2a_0a_2+2a_1a_2+a_1^2+3a_2^2\rangle$$
$$\textbf{where } \langle a_0,a_1,a_2\rangle=t(k).$$

The resulting program is the following:

$$
\begin{aligned}
&d(0)=1\\
&d(1)=2\\
&d(2)=0\\
&d(2k)=a^2+4ab+4bc+6c^2 \qquad \textbf{where } \langle a,b,c\rangle=t(k)\\
&d(2k+1)=2a^2+2b^2+6c^2+4ac+12bc \qquad \textbf{where } \langle a,b,c\rangle=t(k)\\
&t(0)=\langle 1,0,0\rangle\\
&t(2k)=\langle a^2+4bc+2c^2,\ 2ab+2c^2,\ b^2+c^2+2ac+2bc\rangle\\
&\qquad\qquad \textbf{where } \langle a,b,c\rangle=t(k)\\
&t(2k+1)=\langle 4ac+2b^2+4bc+2c^2,\ a^2+4bc+2c^2,\ 2ab+2ac+2bc+b^2+3c^2\rangle\\
&\qquad\qquad \textbf{where } \langle a,b,c\rangle=t(k)
\end{aligned}
$$

Notice that once we computed the expressions for $t(2k)$ and $t(2k+1)$, since they hold for any $k \geqq 0$, we can discard from the program the equations for $t(1), \ldots, t(2r-1)$. This fact is a general property and it could also be applied for the d's values, so that we could have erased the equations for $d(0)$, $d(1)$ and $d(2)$. □

At the end of this section we would like to derive a *for-loop* program equivalent to the general recursive program we have given. This *for-loop* program will be a generalization of the one presented in [34], which had to be proved correct at the expense of many theorems and long proofs and worked only for Fibonacci recurrence relations.

Unfortunately the recursion which occurs in the given general recursive program is not a tail-recursion and therefore its translation into a *for-loop* program is not immediate.

Nevertheless such a translation is possible [31] because of the equivalence between the following recursive schema S and the flowchart in Fig. 5. Suppose we are given the following recursive definition of the function $g(k)$:

$$
g(k)=\begin{cases} e & \textbf{if } \mathrm{zero}(k)\\ a(g(b(k))) & \textbf{if } \mathrm{even}(k)\\ c(g(d(k))) & \textbf{if } \mathrm{odd}(k)\end{cases} \qquad \text{(Schema S)}
$$

where $\mathrm{zero}(k)$ tests whether n is zero, and $\mathrm{even}(k)$ and $\mathrm{odd}(k)$ test whether n is even or odd, respectively.

It can easily be proved by induction on the depth of recursion that $g(k)$ can be computed by the flowchart program (with one stack) given in Fig. 5.

We can apply that result for computing the recursively defined function $t(k)$ which occurs in our program. Since in that program the b and d operations correspond to integer division by 2, the content of the stack at the label L in the given flowchart is the binary expansion of k. We assume that the empty stack represents the binary expansion of 0.

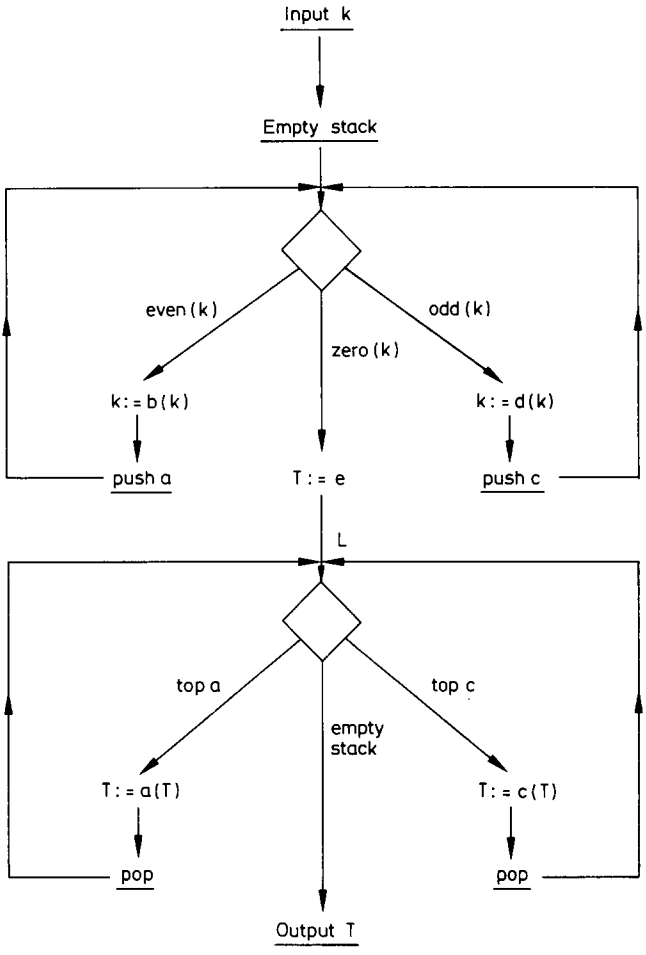


Fig. 5. A flowchart program corresponding to the program schema S.

Therefore we can obtain the following general *for-loop* program for evaluating in logarithmic time any homogeneous linear recurrence relation of order r, as given by the Eqs. 4.1.

for-loop program for evaluating linear recurrence relations (as given by Eqs. 4.1)

$\{b \geqq 0;\ h(0)=h_0;\ \ldots;\ h(r-1)=h_{r-1};\ h(n)=L(h(n-r), \ldots, h(n-1))\quad \text{for } n \geqq r\}$

$$\left.\begin{array}{l}\textbf{if } n=0 \textbf{ then } B(0), l:=0,0\\ \quad\textbf{else begin } m, i, l:=n, 0, \lfloor \log n \rfloor;\\ \quad\quad\textbf{while } i \leqq l \textbf{ do } B(i), m, i:=\mathrm{rem}(m/2), m/2, i+1 \textbf{ od end};\end{array}\right\} \quad (1)$$

$$\left\{n=\sum_{i=0}^{l} B(i)\cdot 2^i \textbf{ and } l=\lfloor \log n \rfloor\right\}$$

$$
\begin{array}{l}
t(0) := \langle 1, 0, \ldots, 0\rangle; \\
\quad\vdots \\
t(r-1) := \langle 0, 0, \ldots, 1\rangle; \\
\textbf{for } i = r \textbf{ to } 2r-1 \textbf{ do } h(i) := L(h(i-r), \ldots, h(i-1)); \\
\qquad\qquad\qquad\qquad t(i) := \langle L(\pi 0(t(i-r)), \ldots, \pi 0(t(i-1))), \ldots, \\
\qquad\qquad\qquad\qquad\qquad L(\pi r-1(t(i-r)), \ldots, \pi r-1(t(i-1)))\rangle \\
\qquad\qquad\qquad \textbf{od};
\end{array}
\qquad (2)
$$

$$
\begin{array}{l}
T := t(0); \\
\textbf{for } p = l \textbf{ downto } 1 \textbf{ do} \\
\quad \textbf{if } B(p) = 0 \textbf{ then} \\
\quad T := \left\langle \sum_{i=0}^{r-1} \pi i(T) \left[\sum_{j=0}^{r-1} \pi j(T) \cdot \pi 0(t(i+j)) \right], \ldots, \right. \\
\qquad \left. \sum_{i=0}^{r-1} \pi i(T) \left[\sum_{j=0}^{r-1} \pi j(T) \cdot \pi r-1(t(i+j)) \right] \right\rangle \\
\quad \textbf{else} \\
\quad T := \left\langle \sum_{i=0}^{r-1} \pi i(T) \left[\sum_{j=0}^{r-1} \pi j(T) \cdot \pi 0(t(i+j+1)) \right], \ldots, \right. \\
\qquad \left. \sum_{i=0}^{r-1} \pi i(T) \left[\sum_{j=0}^{r-1} \pi j(T) \cdot \pi r-1(t(i+j+1)) \right] \right\rangle \\
\qquad\qquad \textbf{od};
\end{array}
\qquad (3)
$$

$$
\begin{array}{l}
\textbf{if } B(0) = 0 \textbf{ then } H := \sum_{i=0}^{r-1} \pi i(T) \left[\sum_{j=0}^{r-1} \pi j(T) \cdot h(i+j) \right] \\
\qquad\qquad \textbf{else } H := \sum_{i=0}^{r-1} \pi i(T) \left[\sum_{j=0}^{r-1} \pi j(T) \cdot h(i+j+1) \right]; \\
\qquad\qquad \{H = h(n)\}
\end{array}
$$

(1) Computation of the binary expansion of n stored in the array $B(i)$ for $i = l, \ldots, 0$.
(2) Computation of the values $h(r), \ldots, h(2r-1)$, $t(r), \ldots, t(2r-1)$.
(3) Computation of T. $T = t(n/2)$ if n is even. $T = t((n-1)/2)$ if n is odd.

The method presented here also allows a fast evaluation of *non-homogeneous* linear recurrence relations, because in that case the solution is the sum of two terms: one corresponding to the associated homogeneous relation, which we can evaluate in logarithmic time, and the other one being a particular solution of the given non-homogeneous relation [22]. The method can also be extended to the case of a set of *mutual depending* linear recurrence relations (with constant coefficients). In fact it is always possible to reduce such a set, via substitutions, to a set of *independent* relations [22] and solve them simultaneously using the tupling strategy. Furthermore one can apply the general algorithms we have given in this section to recurrence relations holding in other algebraic structures, different from the one of the integers with usual addition and multiplication for which we presented it.

The properties of the algebraic structures for which one can use our algorithms are the ones necessary for the validity of the efficiency equation 9′.

They can be derived as follows.

We obviously need to have algebras with two binary operations, say $\cdot$ and $+$, such that $\cdot$ distributes over $+$ on both sides (i.e. ringoids). Since Eq. 9′ should hold for $k=0$, we have:

$$G(a_0, a_1, n)=1 \cdot G(a_0, a_1, n)+0 \cdot G(a_0, a_1, n+1).$$

Therefore we need 0 to be an absorbent element for $\cdot$ and an identity for $+$ and 1 to be an identity element for $\cdot$.

Equation 9′ for $k=1$ does not impose any extra property.
If Eq. 9′ holds for $k=h$ and $k=h+1$, then it would hold also for $k=h+2$ if we assume that associativity and commutativity hold for $+$ and distributivity holds for $\cdot$ over $+$. In fact:

$$\begin{aligned}
&G(a_0, a_1, n+h+2)=G(a_0, a_1, n+h+1)+G(a_0, a_1, n+h)\\
&\quad=(G(1,0,h+1) \cdot G(a_0, a_1, n)+G(0,1,h+1) \cdot G(a_0, a_1, n+1))\\
&\qquad+(G(1,0,h) \cdot G(a_0, a_1, n)+G(0,1,h) \cdot G(a_0, a_1, n+1)) && \text{by 9′}\\
&\quad=(G(1,0,h+1)+G(1,0,h)) \cdot G(a_0, a_1, n)\\
&\qquad+(G(0,1,h+1)+G(0,1,h)) \cdot G(a_0, a_1, n+1) && \text{by hypotheses on the algebra}\\
&\quad=G(1,0,h+2) \cdot G(a_0, a_1, n)+G(0,1,h+2) \cdot G(a_0, a_1, n+1) && \text{by induction.}
\end{aligned}$$

Therefore our algorithms can be used for any *semiring* structure (i.e. an algebra with two binary operations, say $\cdot$ and $+$, such that: $\cdot$ distributes over $+$ on both sides; $+$ is associative and commutative and $\cdot$ is associative) which has an absorbent element for $\cdot$ (i.e. an element 0 s.t. $\forall x\, 0 \cdot x=x \cdot 0=0$) which is also the identity for $+$ (i.e. $\forall x+0=0+x=x$) and an identity element for $\cdot$ (i.e. an element 1 s.t. $\forall x 1 \cdot x=x \cdot 1=x$). For instance our algorithms can be applied to the semiring of truthvalues with *true*, *false*, $\wedge$ and $\vee$.

5. Comparing the Transformation Technique and the Stepwise Refinement Technique

In this section we would like to compare the transformation technique [8] and the stepwise refinement technique [35] making some general comments and, in particular, discussing their features when these techniques are applied for writing programs which evaluate linear recurrence relations. Both techniques indeed have their merits. The stepwise refinement method uses for the solutions of recurrence relations structured concepts, like matrices and binary representations, but it seems to require more creative thoughts than the transformation technique. In particular in the stepwise refinement approach, the programmer has to be familiar and take advantage of the fact that given two matrices $A=\begin{bmatrix} a+b & b \\ b & a \end{bmatrix}$

and $A' = \begin{bmatrix} c+d & d \\ d & c \end{bmatrix}$ we have that: $A \cdot A' = A' \cdot A = \begin{bmatrix} e+f & f \\ f & e \end{bmatrix}$, where e and f depend on a, b, c and d. This is an *a priori* requirement for devising a suitable loop structure and, if the program is not organized with such a loop, it seems very hard, or even impossible, to achieve the desired logarithmic efficiency [18, 30, 34, 27].

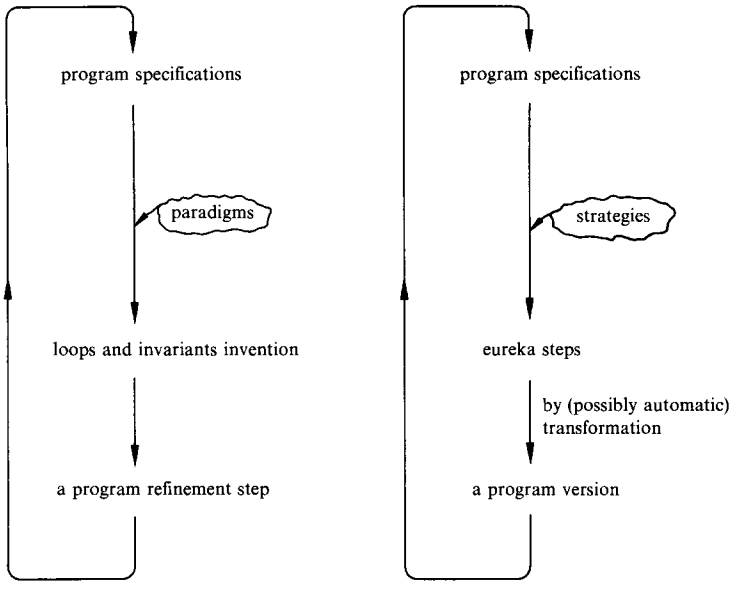
This situation fits very well with the analogy given in [9] by Burstall and Feather, concerning the stepwise refinement technique and the transformation technique, which are respectively compared with sculpture and plasticine modelling. In sculpture, we all know that once the outline of a statue is carved in a rock, there is very little possibility for changing the basic idea of the artistic composition. In our case, using the stepwise refinement technique, the "outline of the statue" consists, in the outermost loop for performing the successive squarings of the matrix $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$.

That loop has to be controlled by the binary representation of the value n for which we would like to compute the solution of the recurrence relation (see [34]). Different approaches to the outermost loop could have made a logarithmic running time almost impossible to be achieved in a simple way.

On the contrary, we showed in this paper that using the transformation technique a very efficient program can be obtained, without a deep knowledge of recurrence relations or matrix theory. The transformation technique, via its strategies, led us to the desired algorithm without great effort.

Obviously in transforming programs we have to make some clever steps, called "eurekas" [8], which are not always easy to make. However, we have strategies, such as composition, tupling, generalization, simplification, etc. [9], which help us to produce eurekas and allow us to improve programs performances. These strategies play the analogous role of the programming paradigms [17] which help the programmer in choosing the suitable loop structures, when using the stepwise refinement technique. The situation of program construction using the two techniques may be roughly depicted as in Fig. 6.

We are not claiming that the transformation method is better than the stepwise refinement method, but we would like to stress that in our examples some very interesting features of the program transformation technique with respect to the stepwise one have been exploited. In particular we have seen that, using standard strategies, it was relatively simple to derive the various "eureka" definitions for our programs, while it seems difficult to directly formulate the complex loop invariant of an iterative program for solving recurrence relations in logarithmic time [34]. In particular the notion of matrix as grouping of values for a fast evaluation of linear recurrence relations has been derived by simple application of the tupling strategy, while that notion is in some sense "primitive" (i.e. not derivable) in the construction of an efficient program using the stepwise refinement technique. It is interesting to notice that, in the derivation of our programs, the simultaneous evaluation of some expressions for avoiding redundant computations, automatically achieved the desired logarithmic performance. One might say that it was the same efficiency requirement which, via the tupling strategy application, "rediscovered" for us the notion of matrix. Indeed it also



(i) The Stepwise Refinement technique (ii) The Transformation technique

Fig. 6. The Program Construction Cycle according to the stepwise refinement technique and the transformation technique.

"rediscovered" the notion of symmetric matrix, as we showed towards the end of Sect. 3.

Another possible advantage of the program transformation approach is the use of the functional style of programming, recently advocated in [3].

In the last part of this section we would like also to stress the strong connection between the tasks of devising loops and finding eurekas, already mentioned in [9]. Related work in this direction has been done in [7]. Obviously the "eureka" steps are "intelligent" steps as the inventions of loops invariants are. For these inventions, creativity and programming experience play a crucial role. For example, the "linear combination eureka" has been as crucial as the idea successive squaring of matrices is for the stepwise refinement technique, but we think that such a eureka is, in some sense, less difficult.

One can often observe an interesting correspondence between the eureka steps and the loops invariants. This correspondence is not claimed in general, but it is valid when the recursive program has a "simple" kind of recursion (which includes tail-recursion) easily translatable into a loop. To clarify the ideas we will now give the following two examples.

Example 4. In [8] the following program for computing the Fibonacci function in linear time is given:

$$\begin{cases} \mathrm{fib}(0)=1 \\ \mathrm{fib}(1)=1 \\ \mathrm{fib}(n)=u+v & \textbf{where } \langle u, v\rangle = g(n-1) \\ \quad g(1)=\langle 1, 1\rangle \\ \quad g(n)=\langle u+v, u\rangle & \textbf{where } \langle u, v\rangle = g(n-1). \end{cases}$$

It is based on the eureka step which consists of defining the function $g(n)=\langle \text{fib}(n), \text{fib}(n-1)\rangle$ for $n \geqq 1$. On the other hand, the iterative program for computing the same function in linear time may be as follows:

$$
\begin{array}{l}
\{n \geqq 0\} \\
\quad i, u, v := -1, 1, 0 \\
\{\langle u, v\rangle = \langle \text{fib}(i+1), \text{fib}(i)\rangle \textbf{ and } i \leqq n-1\} \\
\textbf{while } i < n-1 \textbf{ do} \\
\qquad\qquad t := u+v;\ v := u;\ u := t;\ i := i+1 \\
\qquad\qquad \textbf{od} \\
\{n \geqq 0 \textbf{ and } u = \text{fib}(n)\}.
\end{array}
$$

One can easily see that the two auxiliary variables u and v of the iterative program are equal to the projections of the function g defined in the eureka step. We have in fact: $g(i+1)=\langle u, v\rangle$. □

Example 5. Since $r(n)=\langle G(1,0,n), G(0,1,n)\rangle$ (see Eq. 23) using Eqs. 27 and 28 we have: $r(n)=\langle \text{fib}(n-2), \text{fib}(n-1)\rangle$. Therefore we can transform program P3.1 into the following:

$$
\begin{array}{rll}
\text{fib}(n) = \pi 1(r(n+1)) & & \textit{Program P3.2} \\
r(0) = \langle 1, 0\rangle & & \\
r(2k) = \langle a^2+b^2,\ 2ab+b^2\rangle & \textbf{where } \langle a, b\rangle = r(k) & \\
r(2k+1) = \langle 2ab+b^2,\ (a+b)^2+b^2\rangle & \textbf{where } \langle a, b\rangle = r(k) &
\end{array}
$$

where $\pi 1(\langle e0, e1\rangle) = e1$.

For this recursive program we can easily derive the following iterative version:

$$
\begin{array}{l}
\left\{n \geqq 0 \textbf{ and } l = \lfloor \log(n+1)\rfloor \textbf{ and } n+1 = \sum_{i=0}^{l} B(i)\cdot 2^i\right\} \\
\quad u, v := 1, 0; \\
\quad p := l; \\
I \equiv \left\{\langle u, v\rangle = \langle \text{fib}(m-2), \text{fib}(m-1)\rangle\right. \\
\quad = \langle \text{fib}^2(k-2)+\text{fib}^2(k-1),\ 2\,\text{fib}(k-2)\,\text{fib}(k-1)+\text{fib}^2(k-1)\rangle \qquad \textbf{if } m = 2k \\
\quad = \langle 2\,\text{fib}(k-2)\,\text{fib}(k-1)+\text{fib}^2(k-1),\ (\text{fib}(k-2)+\text{fib}(k-1))^2+\text{fib}^2(k-1)\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } m = 2k+1 \\
\quad \textbf{where if } l = p \textbf{ then } m = 0 \textbf{ else } m = \left.\sum_{i=p+1}^{l} B(i)\cdot 2^{i-p-1}\right\} \\
\quad \textbf{while } p \geqq 0 \textbf{ do if } B(p) = 0 \textbf{ then } u, v := u^2+v^2,\ 2uv+v^2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } u, v := 2uv+v^2,\ (u+v)^2+v^2; \\
\qquad\qquad\qquad p := p-1 \\
\qquad\qquad \textbf{od} \\
\{n \geqq 0 \textbf{ and } v = \text{fib}(n)\}
\end{array}
$$

This program can be derived from the general for-loop program (actually only from the section for computing T) given in Sect. 4 by recalling that:

i) T is equal to $t(n/2)$ if n is even or $t((n-1)/2)$ if n is odd; and

ii) in order to compute $\text{fib}(n)$ we need only to compute the second projection of $t(n+1)$, because in the general for-loop program the function t plays the role of the function r in program P3.2 and in our case $r(n+1)=\langle \text{fib}(n-1), \text{fib}(n)\rangle$. Therefore $t(n+1)$ can be obtained as the value of the variable T in the general

program assuming that the array B contains the binary expansion of $n+1$ and the for-loop is performed for all digits in B.

To complete the derivation of our iterative program from the general one, it is enough to consider that in our case T is $\langle u, v\rangle$ and $t(0)=\langle 1, 0\rangle$, $t(1)=\langle 0, 1\rangle$, $t(2)=\langle 1, 1\rangle$ and $t(3)=\langle 1, 2\rangle$.

The expression for $\langle \mathrm{fib}(m-2), \mathrm{fib}(m-1)\rangle$ in the invariant I can be obtained from the Eq. 23 on which the recursive program P3.2 is based. In fact, since $r(m)=\langle G(1, 0, m), G(0, 1, m)\rangle=\langle \mathrm{fib}(m-2), \mathrm{fib}(m-1)\rangle$ using Eqs. 25 and 26, we have:

$$\begin{aligned}&\langle \mathrm{fib}(m-2), \mathrm{fib}(m-1)\rangle\\ &\quad=\langle \mathrm{fib}^2(k-2)+\mathrm{fib}^2(k-1),\ 2\cdot \mathrm{fib}(k-2)\,\mathrm{fib}(k-1)+\mathrm{fib}^2(k-1)\rangle\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } m=2k\\ &\quad=\langle 2\cdot \mathrm{fib}(k-2)\,\mathrm{fib}(k-1)+\mathrm{fib}^2(k-1),\ (\mathrm{fib}(k-2)+\mathrm{fib}(k-1))^2+\mathrm{fib}^2(k-1)\rangle\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } m=2k+1.\end{aligned}$$ □

6. Conclusions

We derived some algorithms for the evaluation of homogeneous linear recurrence relations with constant coefficients in logarithmic time. We used the program transformation technique and it exhibited very interesting features in deriving those algorithms. Applying known transformation strategies, as the "generalization" and the "tupling" strategy, we were able to obtain in a very simple way efficient programs. In order to use other techniques for constructing programs with analogous performances, one would have had a quite sophisticated knowledge about matrix theory and linear recurrence relations theory. In particular we compared the transformation technique with the stepwise refinement technique and we showed some possible advantages of using the former one. We also presented an algorithm which is a generalization to any homogeneous linear recurrence relation of the one in [34]. The proof of its correctness is implicitly given during its derivation.

Acknowledgements. We are grateful to Professor E.W. Dijkstra for suggesting the problem to us, and to Dr. J. Darlington and Dr. M. Feather for many stimulating conversations. The referees' comments were very useful and perceptive. We also thank the Science Research Council and the Consiglio Nazionale delle Ricerche for their financial support. Many thanks to Miss Kerse for her excellent typing.

References

1. Arsac, J., Kodratoff, Y.: Some techniques for recursion removal from recursive functions. ACM Trans. Progr. Lang. Syst. **4**, 2, 295–322 (1982)
2. Aubin, A.: Mechanizing structural induction: Part I and II. Theor. Comput. Sci. **9**, 3, 329–362 (1979)
3. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM **21**, 8, 613–641 (1978)
4. Bauer, F.L., Broy, M., Partsch, H., Pepper, P., Wössner, H.: Systematics of transformation rules, TUM-INT-BER-77-12-0350. München: Institut für Informatik, Technische Universität, 1978
5. Bauer, F.L., Partsch, H., Pepper, P., Wössner, H.: Notes on the project CIP: outline of a transformation system, TUM-INFO-7729. München: Institut für Informatik, Technische Universität, 1977
6. Bauer, F.L. and Wössner, H.: Algorithmic language and program development. Berlin, Heidelberg, New York: Springer 1981

7. Broy, M., Krieg-Brückner, B.: Derivation of invariant assertions during program development by transformation. ACM Trans. Progr. Lang. Syst. **2**, 3, 321–337 (1980)
8. Burstall, R. M., Darlington, J.: A transformation system for developing recursive programs. JACM **24**, 1, 44–67 (1977)
9. Burstall, R.M., Feather, M.: Program development by transformation: an overview. Proc. 'Les Fondements de la Programmation'. Amirchahy, M., Neel, D. (eds.) Toulouse, pp. 45–55. IRIA-SEFI, France 1977
10. Chatelin, P.: Self-redefinition as a program manipulation strategy. Proc. Symp. Artif. Intellig. Progr. Lang. ACM SIGPLAN Notices and SIGART Newsletter, 174–179 (1977)
11. Cohen, N.H.: Source-to-source improvement of recursive programs. Ph. D. Thesis, Harvard University, TR. 13–80, Aiken Computation Lab., Cambridge, MA, 1980
12. Dahl, O-J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. London: Academic Press 1972
13. Darlington, J., Burstall, R.M.: A system which automatically improves programs. Acta Informat. **6**, 41–60 (1976)
14. Feather, M.S.: ZAP program transformation system: Primer and user manual, DAI Research Report No. 54, Dept. of Artificial Intelligence, University of Edinburgh, 1978
15. Feather, M.S.: A system for developing programs by transformation. Ph. D. Thesis, University of Edinburgh, 1979
16. Feather, M.S.: A system for assisting program transformation. ACM Trans. Progr. Lang. Syst. **4**, 1, 1–20 (1982)
17. Floyd, R.W.: The paradigms of programming. CACM **22**, 8, 455–460 (1979)
18. Gries, D., Levin, G.: Computing Fibonacci Numbers (and similarly defined functions) in log time. Info. Proc. Lett. **10**, 68–75 (1980)
19. Hoggatt, V.E. Jr.: Fibonacci and Lucas Numbers. Boston: Houghton Mifflin Co. 1969
20. Huet, G., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Informat. **11**, 31–55 (1978)
21. Kott, L.: About transformation system: A theoretical study. 3ème Colloque International sur la Programmation, 232–247, Dunod, Paris 1978
22. Liu, C.L.: Introduction to Combinatorial Mathematics. McGraw-Hill 1968
23. Partsch, H., Pepper, P.: Program transformations on different levels of programming, TUM-INFO-7715. München: Institut für Informatik, Technische Universität, 1977
24. Partsch, H., Steinbrüggen, R.: A comprehensive survey on program transformation systems, TUM I8108. München: Institut für Informatik, Technische Universität, 1981
25. Pettorossi, A.: Improving memory utilization in transforming programs. MFCS, Zakopane, Poland. Lecture Notes in Computer Science n. **64**, 416–425. Berlin-Heidelberg-New York: Springer, 1978
26. Pettorossi, A.: Transformation of programs and use of "tupling strategy". Proc. of Informatica '77 Conference, Bled, Yugoslavia, **3–103**, 1–6 (1977)
27. Pettorossi, A.: Derivation of an 0 $(k^2 \log n)$ algorithm for computing order-k Fibonacci numbers from the 0 $(k^3 \log n)$ matrix multiplication method. Info. Proc. Lett. **11**, 4, 5, 172–179 (1980)
28. Schwarz, J.: Verifying the safe use of destructive operations in applicative programs. 3ème Colloque International sur la Programmation, pp. 394–410, Dunod, Paris 1978
29. Steinbrüggen, R.: Equivalent recursive definitions of certain number theoretical functions, TUM-INFO-7714. München: Institut für Informatik, Technische Universität, 1977
30. Urbanek, F.J.: An 0 $(\log n)$ algorithm for computing the nth element of the solution of a difference equation. Info. Proc. Lett. **11**, 2, 66–67 (1980)
31. Walker, S.A. and Strong, H.R.: Characterizations of flowchartable recursions. Proc. 4th ACM Conference on Theory of Computing, Denver, Colorado, pp. 18–34, 1972
32. Wand, M.: Continuation-based program transformation strategies. JACM **27**, 1, 164–180 (1980)
33. Wegbreit, B.: Goal-directed program transformation. IEEE Trans. Software Eng. SE-**2**, 69–79 (1976)
34. Wilson, T.C., Shortt, J.: An 0 $(\log n)$ algorithm for computing general order-k Fibonacci numbers. Info. Proc. Lett. **10**, 2, 68–75 (1980)
35. Wirth, N.: Program development by stepwise refinement. CACM **14**, 4, 221–227 (1971)

Received June 21, 1982