Theoretical Computer Science 53 (1987) 151-167
North-Holland

DERIVATION OF EFFICIENT PROGRAMS FOR COMPUTING SEQUENCES OF ACTIONS*

Alberto PETTOROSSI
Istituto di Analisi dei Sistemi ed Informatica, CNR, 00185 Rome, Italy

Abstract. We consider a class of programs whose output is a sequence of *elementary actions* or *moves*. For that class we provide some transformation strategies for deriving efficient iterative programs with *on-line behaviour*, that is, programs which produce the output moves, one at the time, according to a given sequence ordering.

Using the proposed methods it is possible to answer Hayes' long-standing challenge for deriving a very fast on-line program for the Towers of Hanoi and similarly defined problems.

1. Introduction

When one uses the transformation technique for deriving correct and efficient programs, various strategies may be adopted for the application of the basic transformation rules (see, for instance, [1, 12]).

Those strategies play a crucial role in assuring that the derived programs are equivalent to their corresponding initial versions and that they are also more efficient.

The application of the basic transformation rules, in fact, does not guarantee that equivalence is preserved and efficiency is improved [9].

Unfortunately, a universal strategy which will assure the derivation of efficient equivalent programs cannot exist. That fact comes from a general result valid in logic and proof theory. Indeed, in a sufficiently powerful logical calculus, it is impossible to define a strategy which makes a system automatically prove any given theorm. Nevertheless, one can define various strategies which are successful when the theorems to be proved satisfy some given conditions.

In this paper we will consider a particular class of programs and we will provide a powerful strategy for improving their efficiency. In particular that strategy will achieve the on-line behaviour (which will be formally defined later).

We will write our programs using recursive equations, as the ones presented in [2], and we will restrict our attention to programs which produce strings as output, that is, programs whose outputs belong to monoids.

* A preliminary version of this paper was presented at the 11th Colloquium on Trees in Algebra and Programming (CAAP), in Nice, France, March 1986 with the title "Transformation strategies for deriving on-line programs".

Those programs, which we will call *string-producing programs*, often occur in practical situations. They include all procedures which generate sequences of instructions (or plans) for solving problems and they are often encountered in the area of artificial intelligence, compiler construction, game playing, robotics, etc.

We think that for those programs a nice feature to possess is what we call the *on-line behaviour*, that is, the output string is produced incrementally, one element at the time, in a given order (which is the order in which the consumer reads that string).

Now we present some examples of string-producing programs.

Example 1.1 (*Towers of Hanoi program*). The function $f(n, A, B, C)$ computes the sequence of moves for moving n disks (of different size) which are stacked as a tower on a peg A (with smaller disks on top of larger ones), from peg A to peg B using peg C as an auxiliary peg.

Only one disk at the time can be moved and smaller disks can be placed on top of larger disks only.

The recursive equations for the function f are:

$$f: \text{number} \times \text{Peg}^3 \to \text{Moves}^*,$$

$$f(n+1, A, B, C) = f(n, A, C, B) : AB : f(n, C, B, A),$$

$$f(0, A, B, C) = \text{skip},$$

where $\text{Peg} = \{A, B, C\}$ and Moves* is the monoid freely generated by the set of possible moves $\{AB, BC, CA, BA, CB, AC\}$ with the concatenation operation : and the identity element skip.

Example 1.2 (*Hilbert's curves of order n*). For any given integer n the following function Hilbert(n) describes a curve on a two-dimensional space [8]:

$$\text{Hilbert}: \text{number} \to \text{HMoves}^*,$$

$$\text{Hilbert}\,(n) = A(n),$$

$$\begin{aligned}
A(n+1) &= D(n) : \text{W} : A(n) : \text{S} : A(n) : \text{E} : B(n), \\
B(n+1) &= C(n) : \text{N} : B(n) : \text{E} : B(n) : \text{S} : A(n), \\
C(n+1) &= B(n) : \text{E} : C(n) : \text{N} : C(n) : \text{W} : D(n), \\
D(n+1) &= A(n) : \text{S} : D(n) : \text{W} : D(n) : \text{N} : C(n),
\end{aligned}$$

$$A(0) = B(0) = C(0) = D(0) = \text{skip},$$

where N, S, E, W denote the unit moves towards North, South, East, and West, respectively. HMoves = {N, S, E, W}. For instance, Hilbert(2) is shown in Fig. 1, in which we also indicated the individual moves. For instance the 10th move is a South move, and $B(1)$ is N : E : S.

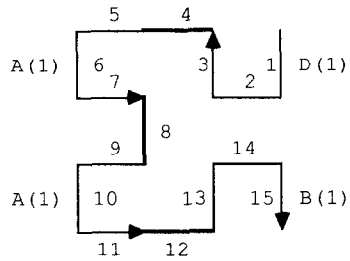

Fig. 1. Example 1.2: Hilbert (2).

Example 1.3 (*N Chinese Rings problem*). It is a generalization of a puzzle considered in artificial intelligence investigations [6]. The objective of the Chinese Rings game is to remove all rings, numbered from 1 to N, from a stick. Ring N can be removed from the stick or put back on it, at any time. Any other ring k may be removed from the stick or put back on it, only if ring $k+1$ is on the stick and rings $k+2$, $k+3, \ldots, N$ are not on the stick.

The following function provides a solution for the game.

$$\text{remove}: \text{rings}^2 \to \text{RMoves}^*,$$

$$\text{put}: \text{rings}^2 \to \text{RMoves}^*,$$

$$\text{remove}(N, N) = N,$$
$$\text{remove}(N-1, N) = N-1 : N,$$
$$\text{remove}(k, N) = \text{remove}(k+2, N) : k : \text{put}(k+2, N) : \text{remove}(k+1, N), \quad \text{if } k \leq N-2,$$

$$\text{put}(N, N) = \underline{N},$$
$$\text{put}(N-1, N) = \underline{N} : \underline{N-1},$$
$$\text{put}(k, N) = \text{put}(k+1, N) : \text{remove}(k+2, N) : \underline{k} : \text{put}(k+2, N), \quad \text{if } k \leq N-2.$$

RMoves* is the monoid freely generated from the set $\text{RMoves} = \{1, \ldots, N, \underline{1}, \ldots, \underline{N}\}$ where k denotes the removal of the ring k, and $\underline{k}$ denotes the putting back of the same ring, for $k = 1, \ldots, N$.

remove(k, N) denotes the sequence of moves for removing from the stick the rings $k, k+1, \ldots, N$, and analogously, put(k, N) denotes the moves for putting back on the stick those rings. (We used k both for denoting the ring k, and its removal from the stick. The reader should distinguish between those two meanings.)

2. On-line computations

In order to motivate our definition of on-line computation, let us start off by recalling Hayes' challenge concerning the Towers of Hanoi problem. In [7], Hayes provides an iterative solution to the problem and he offers as a challenge to derive

it by transformation. Since then, various people (e.g., Er [4] and Walsh [14]) studied that problem (and many variants of it) and they presented recursive and iterative solutions.

We gave a simple transformation derivation of some iterative solutions in [13]. Those solutions are not exactly the ones requested by Hayes. However, they have the same time × space complexity (which is exponential). Therefore, we felt that Hayes' challenge had a satisfactory answer.

However, we also felt that Hayes' program (as well as the programs proposed by Buneman and Levy [3] and Er [4] for similar problems) could claim a little advantage over ours.

Hayes' program, in fact, gives the sequence of moves in exponential time, but it takes only constant time and space to compute the $(k+1)$st move after the computation of the kth one.

The request for that last improvement motivated our investigations. The results we achieved were generalized to the class of string-producing programs and they are presented here.

Let us start with the definition of on-line computations.

Definition. We say that a *computation* which produces a string of l elements is *on-line* if the production of one output element takes *constant time* and *constant space* after the production of the preceding element of the string (before beginning the production of any element, it is possible to have a precomputation phase with time complexity at most proportional to l).

One could also allow for *logarithmic time* and *logarithmic space* w.r.t. the length l of the output string after a precomputation phase proportional to $l \log l$, if the behaviour of the consumer of the output string is not affected by the degradation of performances from constant to logarithmic time × space. Such computations will be called *pseudo on-line.*

According to our definition, the program in [7] determines on-line computations while ours, described in [13], does not.

The on-line property may be required in practice, because the consumer of the output string (e.g. an executing agent) may be asked to work at a convenient speed and to have a small local memory. Indeed, the memory of an executing agent could be much more expensive than the memory of the central computing system.

Obviously, when obtaining the on-line behaviour, we also need to keep the same time × space global performances for the production of the whole sequence of actions.

3. The general problem at schema level

Now we study the problem of transforming recursive programs into on-line iterative versions at schema level.

Let us consider the following recursive schema S, which is general enough to include the above given examples and many others:

$$z(n) = \begin{cases} \langle a_1(z_1, \ldots, z_r), \ldots, a_r(z_1, \ldots, z_r)\rangle, & \text{if } p(n), \\ \langle e_1(n), \ldots, e_r(n)\rangle, & \text{otherwise,} \end{cases}$$

where $\langle z_1, \ldots, z_r\rangle = z(b(n))$. We assume that the function $z(n)$ is an r-tuple of values. The case when $z(n)$ is a single value is obtained for $r = 1$.

Having in S only one nonrecursive clause is not a significant restriction. Everything in what follows can be easily extended to the general case of more than one nonrecursive clause.

For the above schema we assume that

$$\forall i\ 1 \leqslant i \leqslant r\ \forall n\ z_i(n) \equiv a_i(z_1, \ldots, z_r) \in M^*$$

where M^* is a monoid over a given set M.

We also have:

(i) $\forall n\ \exists k \geqslant 0$ s.t. $p(b^k(n)) = \text{false}$, that is, for any n the computation of $z(n)$ terminates;

(ii) $\forall i\ 1 \leqslant i \leqslant r\ e_i(n)$ is an element of M^*, and

$$a_i(z_1, \ldots, z_r) \in (M \cup \{z_1, \ldots, z_r\})^*,$$

that is, $a_i(z_1, \ldots, z_r)$ is the concatenation of its arguments and (possibly) some elements in M.

We also assume that the length of the ith component of $z(n)$ for $1 \leqslant i \leqslant r$ satisfies a linear recurrence relation with constant coefficients. More formally:

$$\forall i\ 1 \leqslant i \leqslant r\ L(z_i(n)) = \sum_{1 \leqslant j \leqslant r} c_{ij} L(z_j(b(n))) + C_i,$$

where the c_{ij} and the C_i are nonnegative integers, and $L: M^* \to \mathbb{N}$ is the length function, which counts the number of elements of M in the given sequence of M^* ($\mathbb{N}$ is the set of natural numbers).

Finally, we assume that the recurrence relations for the $L(z_i(n))$'s can be *inverted*, that is, $\forall n\ \exists d \geqslant 0$ s.t. given the values of $L(z_i(b^j(n)))$ for $1 \leqslant i \leqslant r$ and $0 \leqslant j \leqslant d$ we can compute the values of $L(z_i(b^{j+1}(n)))$ for $1 \leqslant i \leqslant r$.

We will see that the above invertibility hypothesis allows us to efficiently evaluate $z(n)$ using a bounded number of memory cells and avoiding the use of a stack.

The bound d is proportional to the number of cells we need. (In what follows we will give a necessary and sufficient condition for the invertibility property.)

Since there is only one occurrence of the defined function z on the right-hand side of the equations in S, that schema is a linear one. We could then apply the result of Paterson and Hewitt [10] in order to derive an equivalent iterative schema which makes use of a constant number of memory cells only. However, we do not do so because that result does not allow efficient computations: it transforms, in fact, a recursive schema of $O(n)$ steps into an iterative one of $O(n^2)$ steps.

For the schema S we can derive a recurrence relation on the length of the components $z_i(n)$ of $z(n)$. By $L(z(n))$ we denote the r-tuple of those lengths, i.e. $\langle L(z_1(n)), \ldots, L(z_r(n))\rangle$. In what follows, for simplicity, we will write $Lz(n)$ instead of $L(z(n))$, and $Lz_i(n)$ instead of $L(z_i(n))$.

We have $Lz : \mathbb{N} \to \mathbb{N}$ defined by

$$Lz(n) = \begin{cases} \left\langle \sum_{1 \leq j \leq r} c_{1j} Lz_j(b(n)) + C_1, \ldots, \sum_{1 \leq j \leq r} c_{rj} Lz_j(b(n)) + C_r \right\rangle & \text{if } p(n), \\ \langle l_1(n), \ldots, l_r(n)\rangle, & \text{otherwise,} \end{cases}$$

where $\forall i, j\ 1 \leq i, j \leq r$ the values of c_{ij} and C_i can easily be derived from the instance of the schema S under consideration. In particular, $l_i(n) = L(e_i(n))$ for $i = 1, \ldots, r$.

Now let us see how one may compute $z(N)$ for any given N according to the schema S. By hypothesis $\exists k \geq 0$ s.t. $p(b^k(N))$ is false. Let us call s the minimum value of k which satisfies that condition. We may tabulate the values of $Lz(n)$ using the following matrix LZ, called *matrix of lengths*:

	$b^s(N)$	$b^{s-1}(N)$	$\cdots$	$b(N)$	N
Lz_1					
$\vdots$					
Lz_r					

where $Lz_i(b^j(N))$ is the length of the ith projection of z for the input value $b^j(N)$ for $i = 1, \ldots, r$ and $j = s, \ldots, 0$.

Since $p(b^s(N)) =$ false we can fill the leftmost column of LZ. The other columns are filled using the above equations for $Lz(n)$ and the inverse function of b, denoted by b^{-1}. (We assume that such a function exists.)

The value of $LZ(i, n)$, that is, the entry at *row* i and *column labelled by* n in the matrix LZ, gives us the number of elements of M which are in the ith component of $z(n)$. (Indeed $LZ(i, n)$ is another way of writing $Lz_i(n)$.)

Knowing LZ we do not know the value of $z(N)$, but we know the lengths of its components.

Notice that it remains to solve the problem related to the fact that the matrix LZ is unbounded (that is, the value of s depends on the input parameter N). The use, in fact, of an unbounded matrix is equivalent to the use of a stack for evaluating the recursion and, for efficiency reasons, we want to avoid stacks.

In what follows, using the invertibility hypothesis, we will provide a solution for that problem by showing that only a bounded portion of the matrix LZ is needed. For the time being, let us proceed towards the computation of $z(N)$ by assuming that we may access the values in the entire matrix LZ.

The following procedure FIND(i, n, m) solves the problem and it gives us the element of M which is the mth element of the ith component of $z(n)$, denoted by $z_i(n)$, for $1 \leq i \leq r$.

Therefore for any $i = 1, \ldots, r$ and any $m = 1, \ldots, LZ(i, N)$ the mth element of $z_i(N)$ is computed by FIND(i, N, m). For instance, FIND$(i, N, 1)$ produces the first element of $z_i(N)$ and FIND$(i, N, Lz_i(N))$ produces the last element of $z_i(N)$.

In what follows the answer 'no element' means that the output string does not have an mth element. This is the case when $m \leqslant 0$ or $m > Lz_i(N)$.

FIND(i, n, m) =
  if $m \leqslant 0$ or $m > Lz_i(N)$
  then 'no element'
  else
  begin
    let $s_1 : \cdots : s_p$ be $a_i(z_1, \ldots, z_r)$ which is the formal expression of $z_i(n)$ according to the schema S.
    (Thus, $\langle z_1, \ldots, z_r \rangle = z(b(n))$ and $s_k \in (M \cup \{z_1, \ldots, z_r\})$ for $k = 1, \ldots, p$.)
    let for $k = 1, \ldots, p$ l_k be the length of s_k, where the length of any element in M is 1 and the length of z_i for $1 \leqslant i \leqslant r$ is the value of $LZ(i, b(n))$.
    let ls be $[s_1, \ldots, s_p]$ **and** lls be $[l_1, \ldots, l_k]$.
    if $\exists q$ s.t. $\sum_{1 \leqslant i \leqslant q} l_i = m$ and $s_q \in M$ (that is, the mth element of the formal expression of $a_i(z_1, \ldots, z_r)$ is in M)
    then s_q (that is, FIND returns the mth element of $a_i(z_1, \ldots, z_r)$ which is an element of M)
    else FIND(newi, $b(n)$, newm)
    where $\langle$newi, newm$\rangle$ = NEW(ls, lls, m)
  end

NEW(ls, lls, m) =
  if $m \leqslant$ hd(lls)
  then $\langle$newi, $m\rangle$ **where** $z_{\text{newi}} =$ hd(ls)
  else NEW(tl(ls), tl(lls), $m -$ hd(lls))

Notice that the condition on the existence of q is very simple to test. It says that there are two lists lls1 and lls2 s.t. lls1$\langle\rangle$lls2 = lls and the length of lls1 is m. (hd, tl and $\langle\rangle$ denote the usual list functions head, tail and append, respectively.)

We will leave to the reader the correctness proof of the above procedure FIND. The examples in the following section will clarify the ideas.

The procedure FIND always terminates because:

(i) NEW always terminates because $m \leqslant \sum_{1 \leqslant k \leqslant p} l_k = LZ(i, n)$;

(ii) each successive call of FIND decreases the value of its second argument which is bounded from below by $b^s(n)$.

FIND and NEW are tail-recursive and they can easily be transformed into iterative programs.

Using FIND we can compute a generic element of the output string of a string-producing program, if it is possible to translate the given program into an instance

of the schema S. For that transformation it is often crucial to apply the tupling strategy [12].

We can compute the complete output string in any given order of a string-producing program using the following procedure FINDALL. We assume that the output string is $z_i(n)$ and that its length is v. Let $m_1, \ldots, m_v$ be a permutation of the integers: $1, 2, \ldots, v$.

```
FINDALL(i, n, v) =
  j := 1;
  while j ≤ v do print(FIND(i, n, m_j));
                 j := j + 1 od
```

4. Some examples of application of the procedure FIND

Let us compute the 10th move for drawing the Hilbert's curve of order 2, that is, the 10th move of Hilbert(2). It is the South move S, as shown in Fig. 1.

The first step is to transform the program for Hilbert(n), so that it becomes an instance of the schema S.

We can apply the tupling strategy [12] which tells us to group together function calls which have common subcomputations. Since $A(n)$ shares the computation of $B(n-1)$ with $B(n)$ and $C(n)$, and it also shares the computation of $A(n-1)$ with $D(n)$, we define the tuple $z(n) = \langle A(n), B(n), C(n), D(n)\rangle$.

Some results and more details about the application of the tupling strategy can be found in [12]. That strategy essentially goes back to Burstall and Darlington [2], where the authors define the tupled function $\langle \text{fib}(n+1), \text{fib}(n)\rangle$ for developing a fast algorithm for computing the Fibonacci numbers. Indeed fib$(n+1)$ and fib(n) both share the same subcomputation of fib$(n-1)$.

In our case we derive the following program, where by $zi(n)$ we denote the ith projection of $z(n)$:

$$\begin{aligned}&\text{Hilbert}(n) = z1(n),\\ &z(n+1) = \langle z4{:}\text{W}{:}z1{:}\text{S}{:}z1{:}\text{E}{:}z2,\ z3{:}\text{N}{:}z2{:}\text{E}{:}z2{:}\text{S}{:}z1,\\ &\qquad z2{:}\text{E}{:}z3{:}\text{N}{:}z3{:}\text{W}{:}z4,\ z1{:}\text{S}{:}z4{:}\text{W}{:}z4{:}\text{N}{:}z3\rangle,\end{aligned}$$

where $\langle z1, z2, z3, z4\rangle = z(n)$,

$$z(0) = \langle \text{skip}, \text{skip}, \text{skip}, \text{skip}\rangle.$$

The linear recurrence relation on the lengths of the components of $z(n)$ is easily derived from the program above:

$$\begin{aligned}Lz(n+1) = \langle &2Lz1 + Lz2 + Lz4 + 3,\ Lz1 + 2Lz2 + Lz3 + 3,\\ &Lz2 + 2Lz3 + Lz4 + 3,\ Lz1 + Lz3 + 2Lz4 + 3\rangle\end{aligned}$$

where $\langle Lz1, Lz2, Lz3, Lz4\rangle = Lz(n)$,

$$Lz(0) = \langle 0, 0, 0, 0\rangle.$$

In this case $b = \lambda x.x - 1$ and $N = 2$. Since we want to compute the 10th move, $s = 2$. The matrix LZ is:

	0	1	2
$Lz1$	0	3	15
$Lz2$	0	3	15
$Lz3$	0	3	15
$Lz4$	0	3	15

We call FIND(1, 2, 10) because $i = 1$ (that is, Hilbert(n) is the first projection of $z(n)$) and $n = 2$ because we want to compute a move of Hilbert(2). We have:

$$z1(2) = z4(1){:}\mathrm{W}{:}z1(1){:}\mathrm{S}{:}z1(1){:}\mathrm{E}{:}z2(1)$$

and therefore

$$\mathrm{ls} = [z4(1), \mathrm{W}, z1(1), \mathrm{S}, z1(1), \mathrm{E}, z2(1)],$$

$$\mathrm{lls} = [3, 1, 3, 1, 3, 1, 3].$$

Since the 10th element is not an element of {W, S, E, N} because no prefix of lls has its sum equal to 10, we make a recursive call of FIND. For that purpose we compute:

$$\begin{aligned}&\mathrm{NEW}([z4(1), \mathrm{W}, z1(1), \mathrm{S}, z1(1), \mathrm{E}, z2(1)], [3, 1, 3, 1, 3, 1, 3], 10) =\\ &\mathrm{NEW}([\mathrm{W}, z1(1), \mathrm{S}, z1(1), \mathrm{E}, z2(1)], [1, 3, 1, 3, 1, 3], 10-3) = \cdots =\\ &\mathrm{NEW}([z1(1), \mathrm{E}, z2(1)], [3, 1, 3], 2).\end{aligned}$$

Therefore newi = 1 and newm = 2 because hd([$z1(1)$, E, $z2(1)$]) is the first projection of z. We recursively call FIND(1, 1, 2), that is, we look for the second element in $z1(1)$. We then derive:

$$z1(1) = z4(0){:}\mathrm{W}{:}z1(0){:}\mathrm{S}{:}z1(0){:}\mathrm{E}{:}z2(0)$$

with the list of lengths: [0, 1, 0, 1, 0, 1, 0]. Thus, we have that the 10th move of Hilbert(2) is the second move of $z1(1)$ = W:S:E (because $zi(0)$ = skip for $i = 1, \ldots, 4$). That move is S, as expected.

We take a second example of application of the procedure FIND from the Towers of Hanoi problem.

In [13] by applying the tupling strategy we obtained:

$$\begin{aligned}&f(n, A, B, C) = t1(n),\\ &t(n+2) = \langle t1{:}AC{:}t2{:}AB{:}t3{:}CB{:}t1,\ t2{:}BA{:}t3{:}BC{:}t1{:}AC{:}t2,\\ &\qquad t3{:}CB{:}t1{:}CA{:}t2{:}BA{:}t3\rangle,\end{aligned}$$

where $\langle t1, t2, t3\rangle = t(n)$,

$$\begin{aligned}&t(1) = \langle AB, BC, CA\rangle,\\ &t(0) = \langle \mathrm{skip}, \mathrm{skip}, \mathrm{skip}\rangle,\end{aligned}$$

where $t(n) = \langle f(n, A, B, C), f(n, B, C, A), f(n, C, A, B)\rangle$ and we tuple those three functions together because they share the same subcomputations: $f(n-2, A, B, C)$, $f(n-2, B, C, A)$, $f(n-2, C, A, B)$. As usual ti denotes the ith component of t for $i = 1, 2, 3$.

Notice that the recursive equations for $t(n)$ do not fit the schema S because there are two nonrecursive cases: $t(1)$ and $t(0)$. However, the extension of the general method we presented above, is straightforward and it leads to the following solution.

Suppose we want to compute the 13th move of $f(5, A, B, C)$, that is, the 13th move when moving 5 disks from peg A to peg B. The linear recurrence relations of the lengths of the components of $t(n)$ is:

$$Lt(n+2) = \langle 2Lt1 + Lt2 + Lt3 + 3,\ Lt1 + 2Lt2 + Lt3 + 3,\ Lt1 + Lt2 + 2Lt3 + 3\rangle,$$

where $\langle Lt1, Lt2, Lt3\rangle = Lt(n)$,

$$Lt(1) = \langle 1, 1, 1\rangle, \qquad Lt(0) = \langle 0, 0, 0\rangle.$$

In this case $b = \lambda x.x - 2$. Since we want to compute $f(5, A, B, C)$, it turns out that $s = 2$ and $b^s(5) = 1$. It corresponds to the fact that the relevant base case is $t(1)$ and the matrix of lengths, now called LT, has the leftmost column labelled by 1.

In general, if the number of the disks is odd $b^s(N) = 1$, otherwise $b^s(N) = 0$. LT is as follows:

	1	3	5
$Lt1$	1	7	31
$Lt2$	1	7	31
$Lt2$	1	7	31

We call FIND(1, 5, 13). Since $13 < LT(1, 5) = 31$, we compute:

$$t1(5) = t1(3){:}AC{:}t2(3){:}AB{:}t3(3){:}CB{:}t1(3)$$

with the list of lengths: [7, 1, 7, 1, 7, 1, 7]. We have:

$$\text{NEW}([t1(3), AC, t2(3), AB, t3(3), CB, t1(3)], [7, 1, 7, 1, 7, 1, 7], 13)$$

and we obtain:

$$\text{NEW}([t2(3), AB, t3(3), CB, t1(3)], [7, 1, 7, 1, 7], 5).$$

Thus, newi = 2 (because the head of the first argument is $t2(3)$) and newm = 5.

We call FIND(2, 3, 5). Since $5 < LT(2, 3) = 7$, we compute:

$$t2(3) = t2(1){:}BA{:}t3(1){:}BC{:}t1(1){:}AC{:}t2(1)$$

with the list of lengths [1, 1, 1, 1, 1, 1, 1]. Then we call FIND(1, 1, 1) which produces the first element of $t1(1)$ which is AB.

5. Analysis and improvements of the transformation method

The proposed transformation method consists of two phases:

(1) the derivation of a recursive structure for the given program (maybe by the application of the tupling strategy) such that (i) it fits the schema S (or is a straightforward generalization of it), and (ii) the lengths of the components of the output string can be arranged in a matrix, called matrix of lengths (like LT or LZ above);

(2) the computation of the requested mth element of the output string by using a 'decomposition' of m according to the numbers stored in the matrix of lengths, and also by taking into consideration the recursive structure of the program.

Before analyzing the space and time requirements of the algorithms derived from the proposed method, let us first show that there is no need for storing the entire matrix of lengths LT. It is enough to keep the values of a fixed number of its columns only. That fact makes it possible to derive from a recursive program an efficient iterative version which does not simulate the use of a stack. Actually, as shown in [10], any linear recursive schema can be transformed into an iterative one without the use of a stack, but the derived version is in general of $O(n^2)$ time complexity for a recursive schema of $O(n)$. We cannot accept that degradation of performances, and the invertibility hypothesis we made in Section 3 allows us to derive a very efficient iterative algorithm.

If the recurrence relation on the lengths of the output string is invertible, we need to store only d adjacent columns of the matrix LT, instead of the whole matrix (the parameter d was introduced above when defining the invertibility property). For instance, in the case of the Towers of Hanoi the recurrence relations on the lengths can be inverted as follows.

$$Lt1(n+2) = 2Lt1(n) + Lt2(n) + Lt3(n) + 3,$$

$$Lt2(n+2) = 2Lt2(n) + Lt3(n) + Lt1(n) + 3,$$

$$Lt3(n+2) = 2Lt3(n) + Lt1(n) + Lt2(n) + 3.$$

From those equations we get:

$$Lt1(n) = (3Lt1(n+2) - Lt2(n+2) - Lt3(n+2) - 3)/4,$$

$$Lt2(n) = (3Lt2(n+2) - Lt3(n+2) - Lt1(n+2) - 3)/4,$$

$$Lt3(n) = (3Lt3(n+2) - Lt1(n+2) - Lt2(n+2) - 3)/4,$$

and therefore, we can derive the column of LT with label n from the one with label $n+2$.

If the invertibility hypothesis holds, the amount of memory needed for the procedure FIND is *constant*, that is, only a fixed number of columns of the matrix of lengths is necessary and it does not depend on the input parameters. (In the Towers of Hanoi case, one column is enough because the $Lti(n)$'s depend on the $Lti(n+2)$'s only.)

Now we give a necessary and sufficient condition for the invertibility of a system of recurrence relations. Let us consider without loss of generality the case of 2 equations of order 2.

$$x(n) = a_1 x(n-2) + a_2 x(n-1) + a_3 y(n-2) + a_4 y(n-1) + a_0,$$
$$y(n) = b_1 x(n-2) + b_2 x(n-1) + b_3 y(n-2) + b_4 y(n-1) + b_0.$$

We get:

$$a_1 x(n-2) + a_3 y(n-2) = x(n) - a_0 - a_2 x(n-1) - a_4 y(n-1),$$
$$b_1 x(n-2) + b_3 y(n-2) = x(n) - b_0 - b_2 x(n-1) - b_4 y(n-1).$$

This system of equations has a unique solution iff $\det \equiv a_1 b_3 - b_1 a_3 \neq 0$ by the Rouché-Capelli Theorem. Therefore, given $x(n)$, $y(n)$, $x(n-1)$, $y(n-1)$ and the coefficients a_i and b_i, we can univocally derive $x(n-2)$ and $y(n-2)$ iff $\det \neq 0$.

We close this section by giving a complexity analysis of the method we proposed in Section 3 for deriving on-line iterative programs. Our method requires the initialization of the matrix of lengths, or at least a fixed number of its columns, if the invertibility hypothesis holds. The time complexity of this initialization phase if *logarithmic* with the value of the input parameters. (Recall that linear recurrence relations with constant coefficients can be computed in logarithmic time.) The space complexity is *constant*, if the invertibility hypothesis holds, because the tupling strategy requires a fixed (w.r.t. the program structure) number of components.

After the initialization phase, the computation of a particular element of the output string can be done using FIND, independently from the computation of any other 'past' or 'future' element of that string. The number of steps needed for computing FIND(i, n, m) is proportional to the number of columns of the matrix of lengths LZ multiplied by the time required for the computations related to each column (that is, the time required for the computation of q and the pair ⟨newi, newm⟩). The number of columns is proportional to $\log m$ (as it can be seen by solving the recurrence relations on the lengths of the output string). The computation time related to each column is proportional to the number of the components of the expression of $z_i(n)$ in terms of the $z_i(b(n))$'s and the elements of the set M of actions. That number is related to the syntactic structure of the instance of the schema S under consideration and it is constant.

Therefore, we need $O(\log m)$ time to compute the mth move of the output string, and we have derived a program with *pseudo on-line* behaviour. (We did not manage to achieve the *on-line* behaviour.) Our transformation method derives algorithms which produce the elements of the output string *in any order*, while for the on-line behaviour we need only to obtain the elements of the output string independently from the 'future', that is, from the elements which follow in the canonical order: 1st, 2nd, 3rd, ... and so on. Unfortunately, we had to pay for the achieved *two-sided* independence (from the 'future' and the 'past'), because the time required in our method to compute the $(k+1)$st element of the output string, after the computation of the kth element, is logarithmic and not constant.

In the following section we will show how one can improve the performances of the procedure FIND in some particular cases (which includes the Towers of Hanoi problem). We will achieve the on-line behaviour w.r.t. any given order of the elements of the output string. This results provides an answer to Hayes' challenge and also an improvement on his program [7].

6. The answer to Hayes' challenge

By applying the tupling strategy (in a different way w.r.t. the one presented in Section 1) we obtain the following program for the Towers of Hanoi problem:

$$f(n, A, B, C) = t1(n, A, B, C),$$

$$t(n+1, a, b, c) = \langle t1{:}ab{:}t2,\ t3{:}bc{:}t1,\ t2{:}ca{:}t3\rangle$$

where $\langle t1, t2, t3\rangle = t(n, a, c, b)$

$$t(0, a, b, c) = \langle \text{skip}, \text{skip}, \text{skip}\rangle.$$

Given a sequence $s = m_1 \cdots m_p$ of elements in the set Moves $= \{AB, BC, CA, BA, CB, AC\}$, let us denote by $\underline{s}$ the sequence $\underline{m}_1 \cdots \underline{m}_p$ where for any i the move $\underline{m}_i$ is obtained from m_i by interchanging B and C. For instance, $\underline{AB}$ is AC and $\underline{BC}$ is CB. Obviously, if we interchange B and C twice, we obtain again the given sequence. By applying the above notation, we get the following program version:

$$f(n, A, B, C) = t1(n),$$

$$t(n+1) = \langle \underline{t1}{:}ab{:}\underline{t2},\ \underline{t3}{:}bc{:}\underline{t1},\ \underline{t2}{:}ca{:}\underline{t3}\rangle$$

where $\langle t1, t2, t3\rangle = t(n)$,

$$t(0) = \langle \text{skip}, \text{skip}, \text{skip}\rangle.$$

The matrix LT of the lengths of the components of $t(n)$ is:

$n =$	0	1	2	3	4	$\cdots$
$Lt1 = Lt2 = Lt3 =$	0	1	3	7	15	$\cdots$

where for $n = h$ we have $2^h - 1$. Indeed if there are h disks we need $2^h - 1$ moves. It can easily be derived from the following recurrence relations on the lengths of $ti(n)$ for $i = 1, 2, 3$ that

$$Lt(n+1) = \langle Lt1(n) + Lt2(n) + 1,\ Lt3(n) + Lt1(n) + 1,\ Lt2(n) + Lt3(n) + 1\rangle,$$

$$Lt(0) = \langle 0, 0, 0\rangle.$$

The structure of the matrix LT allows us to interpret the computation of the procedure FIND as a walk through a finite automaton. We call it ToH (short for

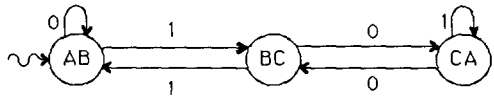

Fig. 2.

Towers of Hanoi) and we represent it as in Fig. 2. The walk is driven by this procedure WALK:

Let $1 \leq m = \sum_{0 \leq i \leq k-1} m_i \cdot 2^i \leq 2^k - 1$.
Use the digits of m, from m_{k-1} to m_0, as follows:
if $m_i = 1$ and $m_{i-1} = \cdots = m_0 = 0$
then STOP
else make the m_i-transition in ToH.
The mth move is the name of the *final* state if we made an even number of transitions, otherwise it is that name with B and C interchanged.

The correctness of WALK is based on the following theorem which shows that, when looking for the mth move of $f(k, A, B, C)$ with $1 \leq m \leq 2^k - 1$, the evaluation of FIND is equivalent to the activation of WALK.

Theorem 6.1. *The mth move, for* $1 \leq m \leq 2^k - 1$, *of* $f(k, A, B, C)$, $f(k, B, C, A)$, *and* $f(k, C, A, B)$ *respectively, is computed by the procedure* WALK *on the automaton* ToH *with initial state AB, BC, and CA, respectively.*

Proof. By induction on k.

Base: $k = 0$. Obvious.

Step. Suppose the theorem true for $k = 0, \ldots, n$. Let us show it for $k = n+1$. Let m be denoted by the digits $m_n \cdots m_0$. There are two cases:

Case 1: $m = 2^n$. By definition the 2^n-th move of $ti(n+1)$, which is a string of length $2^{n+1} - 1$, is the middle move, because $Lti(n) = 2^n - 1$ for $i = 1, 2, 3$ (see the matrix of lengths). Therefore, that move is AB, BC, CA for $i = 1, 2, 3$ respectively. Using the procedure WALK the final state is the initial state because no transitions are made. Thus the mth move is AB, BC, CA, respectively.

Case 2: $m \neq 2^n$. Suppose that the initial state is AB, that is, we want to compute the mth move of $f(k, A, B, C)$. (Analogous proof is for the initial state BC or CA.) Since $Lt1(n) = Lt2(n) = 2^n - 1$ and $Lt1(n+1) = 2^{n+1} - 1$, the mth move of $t1(n+1) = \underline{t1(n)}{:}AB{:}\underline{t2(n)}$ is the $(m - m_n \cdot 2^n)$th of $\underline{t1(n)}$ (or $\underline{t2(n)}$) if $m_n = 0$ (or 1). Using WALK, the mth move is the $(m - m_n \cdot 2^n)$th move computed from the initial state AB (or BC) if $m_n = 0$ (or 1) because a 0-transition (or 1-transition) has been made. For each transition made B and C are interchanged, as required by $\underline{t1(n)}$ (or $\underline{t2(n)}$). By induction hypothesis the theorem is proved. □

Example 6.2. Finding the 44th move of $f(6, A, B, C)$. $44 = 101100$. The state transitions are:

$$AB \xrightarrow{1} BC \xrightarrow{0} CA \xrightarrow{1} CA$$

In this last state the digits to be read are '100'. Therefore CA is the final state. Three transitions are made. The 44th move is: $\underline{CA} = BA$. (The exchange of B and C is due to the fact that the number of transitions is odd.)

We can store the automaton ToH in the following matrix A:

input		0	1
state	AB	AB	BC
	BC	CA	AB
	CA	BC	CA

and therefore we use *constant space* only. The procedure WALK for the initial state AB becomes:

$\{m = \sum_{0 \leq l \leq k-1} m_l \cdot 2^l\}$
$p := k - 1$; state $:= AB$;
while $p \geq 0$ and $\sum_{0 \leq l \leq p} m_l \cdot 2^l \neq 2^p$
do state $:= A$[state, m_p];
$p := p - 1$ **od**;
if even$(k - 1 - p)$
then print(state)
else print(state$[B \leftrightarrow C]$).
{the printed state is the mth move of $f(k, A, B, C)$}

$B \leftrightarrow C$ denotes the interchange of B and C.

Notice that the number of transitions is $k - 1 - p$, because their number $+p$ is the invariant value $k - 1$. In fact, at each new transition, p is decreased by 1 and it was initialized to $k - 1$.

The time necessary to compute the mth move is logarithmic w.r.t. the binary expansion of m. However, we can say that the mth move is computed in *constant time* in the sense of Hayes, because in [7] arithmetical operations (which are logarithmic w.r.t. the binary expansions of their operands) are considered to take constant time as well.

The complete program for computing the moves of the Towers of Hanoi problem with N disks is a loop which invokes the procedure WALK for $m = 1, \ldots, 2^N - 1$. We leave to the reader some minor improvements one can realize to that program, when it is known that the moves are required in the canonical order: 1st move, 2nd move, etc. That program is an improvement on Hayes' iterative algorithm [7] because it has an on-line behaviour for any required ordering of the moves (not only for the canonical one).

The algorithm in [11] also has the on-line behaviour w.r.t. any ordering, but it does not give us for any move m the destination of the disk to be moved. That algorithm uses the least significant part of the binary expansion of m, while ours uses the most significant one and it produces source and destination of the mth move.

7. Conclusions

We proposed a new transformation method for a class of programs which produce strings as output. We presented an algorithm for obtaining equivalent iterative programs which are very efficient and exhibit *on-line behaviour*. By that notion we mean that any element of the output string can be computed very efficiently (for instance, in constant time and space) after the computation of the preceding elements.

Among other examples we applied the proposed method to the Towers of Hanoi problem, providing an answer to a long-standing challenge [7] for deriving a fast on-line iterative program by transformation techniques.

Acknowledgments

Many thanks to L. Meertens, H. Partsch and A. Arnold for their illuminating suggestions. P. Franchi-Zannettacci gave me the enthusiasm for revising a previous version of this paper. A. Labella and my colleagues at IASI provided a nice research environment.

The Italian National Research Council of Italy financially supported this study

References

[1] R.S. Bird, The promotion and accumulation strategies in transformational programming, *ACM TOPLAS* **6**(4) (1984) 487-504.

[2] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *ACM* **24** (1) (1977) 44-67.

[3] P. Buneman and L. Levy, The Towers of Hanoi problem, *Inform. Process. Lett.* **10** (1980) 243-24

[4] M.C. Er, An iterative solution of the generalized Towers of Hanoi problem, *BIT* **23** (1983) 295-30

[5] M.C. Er, An iterative algorithm of the cyclic Towers of Hanoi problem, *Internat. J. Comput. Inform Sci.* **13** (2) (1984).

[6] G.W. Ernst and M.M. Goldstein, Mechanical discovery of classes of problem-solving strategies, *ACM* **29** (1) (1982) 1-23.

[7] P.J. Hayes, A note on the Towers of Hanoi problem, *Comput. J.* **20** (1977) 282-302.

[8] D. Hilbert, Über stetige Abbildung einer Linie auf ein Flächenstück, *Math. Ann.* **38** (1891) 459-46

[9] L Kott, About transformation system: A theoretical study, in: *Proceedings 3ème Colloque International sur la Programmation* (Dunod, Paris, 1978) 232-247.

[10] M.S. Paterson and C.E. Hewitt, Comparative schematology, in: *Proceedings Conference on Concurrent Systems and Parallel Computation*, Project MAC, Woods Hole, MA (1970) 119-127.

[11] H. Partsch and P. Pepper, A family of rules for recursion removal, *Inform. Process. Lett.* **5** (6) (197 174-177.

[12] A. Pettorossi, A powerful strategy for deriving efficient programs by transformation, in: *Proceedings ACM Symposium on Lisp and Functional Programming*, Austin, TX (1984), 273-281.

[13] A. Pettorossi, Towers of Hanoi problems: Deriving iterative solutions by program transformation, *BIT* **25** (1985) 327-334.

[14] T.R. Walsh, Iteration strikes back—at the cyclic Towers of Hanoi, *Inform. Process. Lett.* **16** (1983) 91-93.