数理解析研究所講究録
第339巻 1978年 13-36

E.G & M.T Sep. 7, 1977

MTAC - Mathematical Tabulative Automatic Computing

by Eiichi Goto and Motoaki Terashima

Department of Information Science, Faculty of Science
University of Tokyo, Tokyo, 113, Japan

Abstract

Tabulation vs. recomputation of mathematical function is a typical space vs. time trade off problem in computing. Two principles, (P1) on-demand tabulation and (P2) reclaimable tabulation are proposed to widen the range of applicability of tabulation. For some cases these principles are shown to be similar in effect to recursion elimination. The results of software implementation of these principles are given. Another MTAC - Mathematical Tabulative Architecture for Computers relating to hardware hashing and to a modified buffer (cache) register is also discussed.

Key Words and Phrases : tabulation, recursion elimination, hashing hardware hashing, buffer (cache) register.

14

E.G & M.T Sep. 7, 1977

1. Introduction

The choice between tabulation and recomputation of mathematical functions is an old and typical time vs. space trade off problem and it is an important subject for speeding up computing.

In the old days of hand computing, mathematical tables, either in published forms or written on sheets of paper for the later use by the human computer, were of great value. Since the advent of electronic computers giving a phenomenal increase in computing power, such mathematical tables have been greatly devaluated, as symbolized in the changing of the name of a mathematical journal from MTAC (Mathematical Tables and other Aids to Computation) into MC (Mathematics of Computation) in 1960.

"Data base" may be regarded as functions which can be defined only in terms of very large tables, of great value. In this paper, we shall focus our attention around functions which can be defined in terms of concise mathematical algorithms and will not touch upon data bases.

Recomputation of functions was specially fashionable in the early days of electronic computers, because of severe limitations on the capacity of high speed memories. The continuing trends of increase in the cost effectiveness of high speed memories, however, have demanded and will continue to demand for revaluations of tabulative computing methods. For example, contemporary FFT programs would initially tabulate the values of trigonometric functions to be used repeatedly in the FFT algorithm, which might had to be programmed differently in the early day machines with very small memories.

The main theme of this paper consists in two tabulation principles, "on-demand" and "reclaimable" tabulation, and the aims and scopes of this paper may be summarized into two paraphrased MTAC's.

2. On-Demand and Reclaimable Tabulation

The method as use in FFT may be call "explicit pre-tabulation", since the tabulation is made in advance by explicit programming. This simple and straightforward method, however, may not be applicable in the following cases:

(C1) The values of the arguments of functions for which the functions are (likely) to be used are difficult to foresee.

(C2) The total amount of space needed for tabulating functions exceeds the memory capacity available.

In case of FFT, one would tabulate $\sin(2\pi n/N)$ and $\cos(2\pi n/N)$ for $0 \leq n < N$ (N is a constant). In this case, the space needed would not be large because the table is one dimensional. Moreover all the values of n ($0 \leq n < N$) are known to be used repeatedly.

In case of a two dimensional table, such as in the case of the binomial coefficients $C_{n\ m}$, the space needed would be larger (because it is two dimensional, case (C1)); and it would be often difficult to foresee the maximum range of n and m. The situation would be rather serious for the 6 dimensional Racah coefficients $R(j_1, j_2, j_3, j_4, j_5, j_6)$ also called the 6-J symbol. The situation would be even worse for the 9-J symbol $X(j_1, j_2, \ldots j_9)$. (These symbols are used in quantumn mechanical calculations. cf. [10] for their definitions.)

In such cases, (C1) and (C2), the following tabulation principles, (P1) and (P2) would be helpful.

(P1) On-Demand Tabulation

When the evaluation of a function, f(n,m) say, is invoked, the table entry corresponding to the function f and the values of arguments n and m is checked for its existence first.

If "not-existing" (the entry maybe tagged "non-existing"), then the program for computing f(n,m) is called and a new table entry is created (and maybe detagged) for later retrieval, else the value is retrieved from the table without recomputing f(n,m). Note that wasteful computations of table entries which are never used, the trouble of case (C1), are clearly avoided by applying principle (P1).

(P2) Reclaimable Tabulation

While (P1) is applicable independently by itself, (P2) is always to be applied together with (P1). The basic idea of (P2) is to reclaim (an) table entry(ies) when the tabulation runs out of space, there by changing the reclaimed entry(ies) into "non-existing".

The probability of needs for recomputation and thence the overall efficiency of the tabulation scheme would largely depend upon the reclaiming strategy as to be discussed in section 4. Nevertheless, note that the computation would proceed correctly by virtue of (P1) independently of the reclaiming strategy.

3. Speeding Up Recursion by On-Demand Tabulation

To iterate is human, to recurse is divine. - L. P. Deutsch.

Recursive programs are often regarded as being better structured than iterative programs. Especially, for functions which satisfy recurrence relations, the correctness of recursive programs are often demonstratively obvious as in the following examples, in which n and m are integers.

Gamma function, $\Gamma(1)=1$, $\Gamma(n)=(n-1)\Gamma(n-1)$ for $n\geq 2$:

$\Gamma(n)$ <= (if n=1 then 1 else (n-1)Γ(n-1) fi).

Fibonacci number, $b(1)=b(2)=1$, $b(n) = b(n-1)+b(n-2)$ for $n\geq 3$:

b(n) <= (if n≤2 then 1 else b(n-1)+b(n-2) fi).

Binomial coefficient, $c(n,m) = C^n_m$

$C^n_0 = C^n_n = 1$, $C^n_m = C^{n-1}_m + C^{n-1}_{m-1}$ for $0<m<n$:

c(n,m) <= (if m=0∨m=n then 1 else c(n-1,m) + c(n-1,m-1) fi).

The function nth to extract the n-th element from a linked list (as in LISP) $r = (r_0, r_1, \ldots r_n)$:

nth(n,r) <= (if n=0 then car(r) else nth(n-1,cdr(r)) fi).

However recursive programs are known to have a serious problem. Namely, the speed may be extremely slow due to the repeated recomputations of the same function for the same value(s) of argument(s). In order to elucidate this point, we make use of a graph to be called the "Reference directed graph" or "R-graph" for short, of the computational procedure. Each call of (or reference to) a function is represented as an arrow (directed edge) in the the R-graph. Fig. 1 shows the R-graphs for the computations of Γ(4), b(5), c(4,2) and nth(3,r).

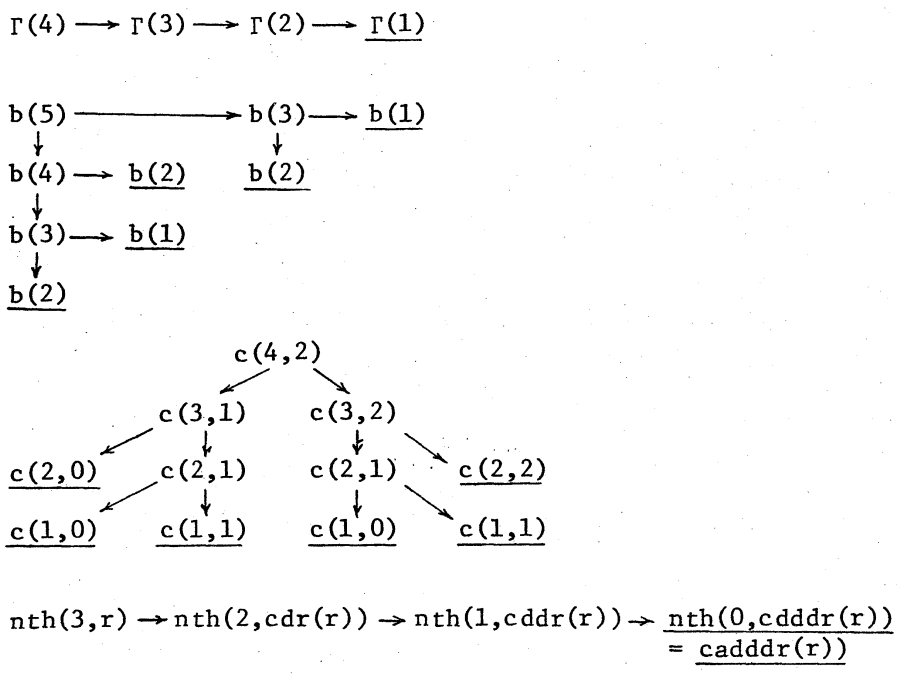

Fig. 1. R-graphs (R-trees) for Γ(4), b(5), c(4,2) and nth(3,r). Leaves of trees are underscored.

It should be noted that the termination proof of an algorithm (i.e., a computational procedure which terminates) is equivalent to showing that the R-graph is a tree of finite order. Hence, the R-graph of an algorithm is necessarily a tree, to be called the R-tree hereinafter.

In usual recursive programming systems†, a traverse of the entire R-tree would be practiced during the computation. Therefore, in the respective computations of b(5) and c(4,2), b(3) and c(2,1) would be computed twice as to be seen from R-trees in Fig. 1. In the recursive computation of b(n) for large n, the number of vertices in the R-tree would explode exponentially, thereby making the computation impracticable.

Foot note. LISP and ALGOL are examples of "usual" systems. The situation may be different in "unusual" systems such as the lazy evaluator [19].

It would be reasonable to use the total number of arrows (i.e., references) in the R-tree as the time cost function of the computation. The respective costs of $\Gamma(n)$, $b(n)$, $c(n,m)$ and $nth(n,r)$ are $n=O(n)$, $2b(n+1)-1 = O(1.618^n)$, $2c(n,m)-1 \leq 2c(n,m/2)-1 = O(2^{n/\sqrt{n}})$ and $n=O(n)$, in which b and c blow up exponentially.

By applying the on-demand tabulation principle (P1) to these recursive computations, all wasteful recomputations would be replaced by table look-ups.

Let the contracted R-graph be the graph obtained by contracting the vertices of the R-graph for the same function for the same value(s) of argument(s) into a single vertex. Specially, let the contracted R-graph of an algorithm be call the R-dag ("dag" stands for "directed acyclic graph". It can be easily proved that the contraction of an R-tree results into a dag). For simplicity for the time being, let us assume that the tabulation space is big enough so that there is no need to invoke the reclaimable tabulation principle (P2). A recursive program with on-demand tabulation would make a traverse through the R-dag instead of through the R-tree. Fig. 2 shows examples of R-dags. For later use, the superfixed ordinals in Fig. 2 show a TS (Topological Sorted)-sequence for each R-dag. We define TS-sequence of an R-dag as a duplication free sequence of all vertices $(v_1, v_2, \ldots v_n)$ such that whenever $v_i \rightarrow v_j$, $i>j$ holds. Such a sequence exists for any dag and the algorithm for finding it is called topological sorting [15, pp.259-265]. For $\Gamma(n)$ and $b(n)$, the TS-sequence is unique. For $c(4,2)$, there are two TS-sequences $(c(2,1),c(3,2),c(3,1),c(4,2))$ and $(c(2,1),c(3,1),c(3,2),c(4,2))$. Only the former is shown in Fig. 2.

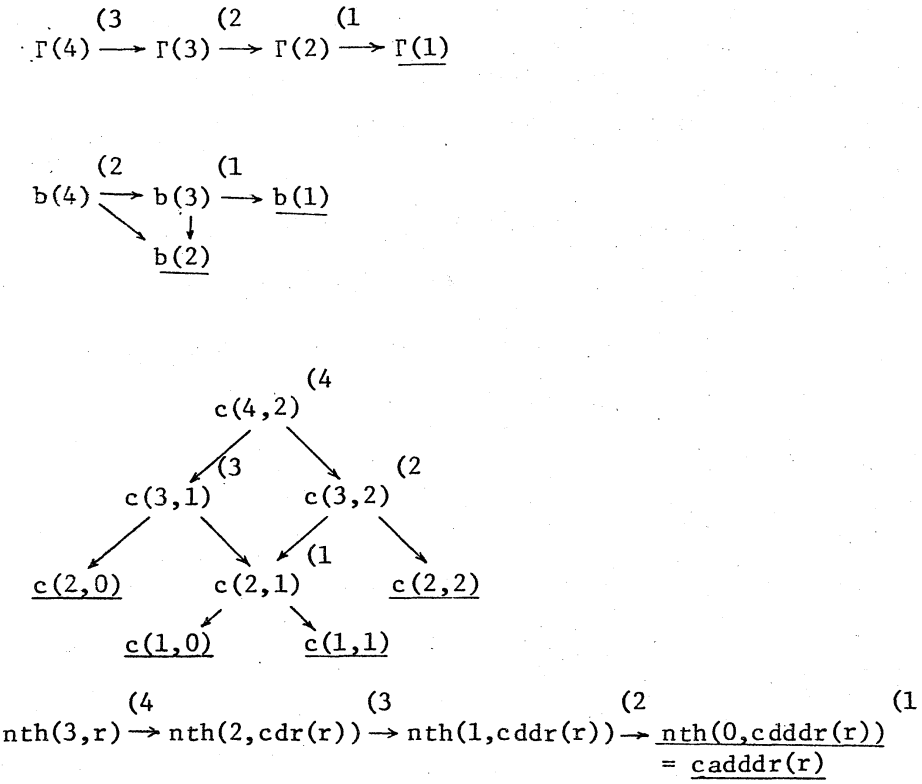

Fig. 2. R-dags for Γ(4), b(4), c(4,2) and nth(3,r)
Leaves are underscored. Superscript ordinals show a Topological Sorted sequence of vertices.

A recursive procedure may be cyclic, i.e., the procedure defining a function makes use of the value of the same function for the same value(s) of argument(s) in the definition itself. The contracted R-graph of such procedure would contain a cycle and the procedure would never terminate. Fig. 3 shows an example.

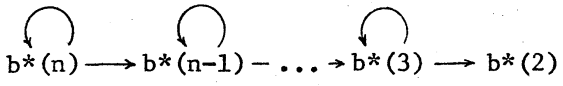

Fig. 3. Vicious Cycles in the Contracted R-grahp of the procedure:
b*(n) <= (if n<2 then 1 else b*(n)+b*(n-1) fi).

E.G & M.T Sep. 7, 1977

A run time "cyclicity test feature" can be easily added to the on-demand tabulation scheme. Namely, besides the two attributes "existence" and "non-existence" of each table entry, "gray" is to be added as the third. "Gray" is to mean that the entry has been entered but the value has not been tabulated yet. Entering a "gray" entry clearly implies a "vicious cycle", or more algorithmically:

```
f(n) <= (begin (if the entry is "gray" then
                    print error message for VICIOUS CYCLE fi);
         (if the entry is "non-existing" then
             make a "gray" entry fi);
         compute and tabulate f(n) making the entry "existing"
         end).
```

It would be reasonable to use the number of vertices and leaves of the R-dag as the space cost function for the on-demand tabulative computation.

Under the assumption of O(1) time (complexity) for table retrieval (implementations are discussed in section 4) it would be reasonable to use the number of arrows in the R-dag as the time cost function for the initial on-demand tabulative computation. Time costs of $\Gamma(n)$, b(n), c(m,n) and nth(n,r) now become $n=O(n)$, $2n-1=O(n)$, about $n^2/2=O(n^2)$ for $m=n/2$ and $n=O(n)$. The time costs for repeated computations are all O(1) by virture of tabulation. The respective space costs for these four functions are $n=O(n)$, $n=O(n)$, about $n^2/4 = O(n^2)$ and $n=O(n)$. These cost are summarized in the following table where "#↑" and "#." respectively means the number of arrows and the number of vertices including leaves.

Function	Time Cost Pure Recursion	Time Cost On-Demand Tabulation Initial	Time cost On-Demand Tabulation Repeated	Space Cost for Tabulation
f(x)	$O(\#\uparrow(\text{R-tree}))$	$O(\#\uparrow(\text{R-dag}))$	$O(1)$	$O(\#.(\text{R-dag}))$
$\Gamma(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
b(n)	$O(1.618^n)$	$O(n)$	$O(1)$	$O(n)$
c(n,m)	$O(2^n/\sqrt{n})$	$O(n^2)$	$O(1)$	$O(n^2)$
nth(n,r)	$O(n)$	$O(n)$	$O(1)$	$O(n)$
m(1,n,r)	$O(4^n/\sqrt{n})$	$O(n^3)$	$O(1)$	$O(n^2)$

Table 1. Time and Space Costs

As another example, take the minimum number of multiplications needed to multiply n matrices $M_1, M_2, \ldots M_n$ where M_i has r_{i-1} rows and r_i columns. It is assumed that pqr multiplications are needed to multiply a p X q matrix by a q X r matrix [1]. Let m(i,j,r) be the minimum number of multiplications for multiplying matrices $M_i, M_{i+1}, \ldots M_j$. m(i,j,r) can be computed recursively as:

m(i,j,r) ⇐ (<u>if</u> i≥j <u>then</u> 0 <u>else</u>

$$\min_{i<k<j} (m(i,k,r) + r_{i-1} r_k r_j + m(k+1,j,r))\ \underline{fi})$$

where r is a list $r = (r_0, r_1, r_2, \ldots r_n)$ and r_i, $0 \le i \le n$, is to be computed by the function nth.

Fig. 4. shows the R-dag for m(1,4,r).

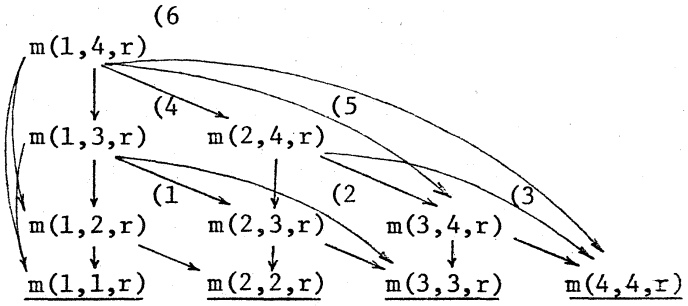

Fig. 4. R-dag for m(1,4,r)

The costs for m(i,j,r) is also given in Table 1, wherein the subfunction nth is also to be subjected to tabulation for speed up.

Recursion elimination, or transformation of recursion(s) into iteration(s) to be more specific, is a practice used for speeding up recursion. In a recursion elimination method, called dynamic programming by some authors [1, pp.67-69], a TS-sequence is generated and traversed by iterative loop(s). Hence, the present authors would use the term "TS-method". For example, the TS-sequence, $\Gamma(i)$, i=2,3, ... n, of $\Gamma(n)$ can be generated by a "for loop" as in:

$\Gamma(n)$ <= (begin m<=1; for i<=2 step 1 until i=n

(do m <= m*(i-1) od); return m as the result end).

The similar can be done for b(n), c(n,m) and m(i,j,r). Specially m(i,j,r) is treated in [1] as an example of dynamic programming rather than of a recursion elimination.

This method is not applicable when it is difficult to generate the TS-sequence from the value(s) of argument(s) given at the root of the R-dag. For example, take the recursive program for taking the n-th element nth(n,r). The unique first vertex of the TS-sequence in this case is nth(0,cd...dr(r)). But this means that the shortend list, which is almost the result itself, is needed to identify the first vertex. Hence, the TS-method obviously fails in this case. Nevertheless, the recursive nth can be easily transformed into an iterative form as:

nth(n,r) <= (while n>0 (do n<=n-1; r<=cdr(r) od);

return car(r) as the result elihw),

by noticing the fact that recursive call on nth is of a special type which actually does not need the control to be returned to the calling site. The lack of general methods applicable to all cases seems to be the fundamental problem of recursion elimination schemes [3,4].

Even when recursion elimination is possible, the resultant iterative program would still have to make references to all arrows of the R-dag and the cost would also be O(#↑(R-dag)), which is the same in order of magnitude (time complexity) as that of the initial computation of recursion with on-demand tabulation†. The actual timing figures would naturally depend upon the software and hardware in use. Nevertheless, it would be worth noting that a tabulative recursive program in some cases can be actually faster than its recursion eliminated iterative version.

Let us assume that Γ(1), Γ(2), ... Γ(N) are used repeatedly with on-demand tabulation in a part of computation. Both recursive and iterative programs would work equally well for this part. But when another part is entered requiring Γ(N+1), the recursive program would compute Γ(N+1) in time O(1) by making use of the tabulated value of Γ(N). The iterative version however, would take time O(N) because it would compute Γ(N+1) from scratch. Hence, the recursive version would run faster beyond same value of N.

Foot note. An algorithm may be improved by skipping references to some vertices. For example, computation of Fibonacci number b(n) can be made faster by using the relation

$b(2n) = b(n)^2 + 2b(n)b(n-1)$,

$b(2n-1) = (b(n)+2b(n-1))b(n-1) - (-1)^n$. It would be natural to regard algorithms, which makes use of such other special relations, as those based on a different principle.

4. Software Implementations and Reclaiming Strategies

On-demand tabulation of a function f(n), say may be programmed explicitly. Let t(n) be an array to be used for tabulating f(n), and let f*(n) be an algorithm for computing f(n) without using t(n) (it may make use of other entries t(n') n≠n'). The program may be like:

f(n) <= (begin (if not_exist(t(n)) then t(n) <= f*(n) fi);
return f(n) as the result end)

More specifically for b(n), let t(n) be initialized as t(1)=t(2)=1 and t(n)=0 for n≥3 which is to be used as a tag (reserved value) for "non-existence". We may write:

b(n) <= (begin r<=t(n); (if r=0 then t(n) <= (r <= f(n-1)+f(n-2) fi);
return r as the result end).

Such "explicit on-demand tabulation" may be used when the memory space is plentiful and there is no need to invoke reclaimable tabulation (P2) and when a simply indexed dense array can be used for the table.

However, when reclaimable tabulation (P2) is needed some storage allocation algorithm has to be implemented and when the tabulation needs some other programming techniques such as the handling of very sparse arrays etc., it would be difficult for the general users to explicitly write such highly specialized programs. For such cases the best would be to provide the users with an "implicit" tabulation package written by specialists.

As an user interface for implicit tabulation, switch statement such as "ON TAB Γ, b, c" and "OFF TAB Γ, b, c" seem to be simple but yet powerful enough to handle situations like selective tabulation. Suppose a function exp, say is used at different places in a program as "... exp(float(n)) ... exp(x) ..." and suppose exp is likely to be

used repeatedly for the same integer values of n, but unlikely for the same real values of x. It would be natural to selectively tabulate only the former. This can be done either by suitably switching on and off "TAB", or by replacing the former by a new function, tabexp(n) = exp(float(n)) and specifying "ON TAB tabexp".

We now discuss the implementation of implicit tabulation which is and should be kept to a large extent transparent to the users. It would be reasonable to use a content addressed table in which the ordered set (tuple) of the function identifier and the value(s) of argument(s) is to be used as the key (content addressing) part (field) of each table entry. Since IC chips for content addressed associative memory are still rather expensive, we may have to use their software version - hash tables - for the time being. Since hashing has been discussed elsewhere [e.g., 16], we only give a summary: while the IC chips just mentioned can retrieve an entry in strict O(1) time independently of the size of the table, software hashing does the similar in statistically expected O(1) time. It may be worth mentioning that a method for handling variable (long) length keys different from those disclosed in Gries [9, pp.227-228] and Knuth [16, p.549] is given in [7,8].

Reclaimable tabulation and "on-demand paging virtual memory" are similar in many aspects. The only difference consists in that while reclaimed entries are simply eliminated and subjected to recomputation upon later use in the former, they are swapped out from the main memory to the secondary memory and swapped in upon later use in the latter. For simplicity, the possible combinations of virtual memory with reclaimable tabulation will not be discussed.

"LRU (Least Recently Used) reclaiming", which is practiced in

virtual memories seems to be a better strategy than "random reclaiming". Let P be the probability of an entry for f(n,m), say is remaining in the table. Then the expected cost for computing f(n,m) is $C = P + C_R(1-P)$ where C_R and $C_R(1-P)$ are the recomputing and expected recomputing time costs, respectively. In order to keep the latter below a constant K of the order on unity, i.e., $C_R(1-P) \leq K = O(1)$, P must satisfy $P \geq 1 - K/C_R$. Since tabulation is expected to be most effective when C_R is large, this implies P must be kept close to unity. While this is difficult in "random reclaiming", P=1 will be maintained for non-obsolete more or less (not the least) recently used entries in "LRU reclaiming". "Reclaim all when (the table becomes) full", a very simple strategy would be more practical than "LRU", since the software overheads, in both space and time for strict LRU could be rather high. ("LRU" can be implemented in O(1) time by using doubly linked list [15, p.451]. The situation will change if hardware LRU units be used.) This would work in the same way as the "LRU" strategy until the table gets full. The tabulation must be initiated from scratch when the reclaiming does take place. Nevertheless, this strategy has yielded satisfactory results in many practical cases. On-demand tabulation was implemented in a system called HLISP as an additional feature to LISP [6,8,13,22]. Control statements of the forms "ASSOCCOMP† (<list of function names>)" and "UNASSOCCOMP (<list of function names>)" are used instead of "ON TAB" and "OFF TAB".

Foot note. This stands for "associative computation".
A more specific term "tabulative computing" may be better.

As an example, the recursive evaluation of Fibonacci number b(21) = 10946, becomes about 1000 times faster upon the initial application of tabulation and about 30000 times faster when it is used frequently.

The "cyclicity test feature" was also implemented in HLISP and it has been proved to be a useful debugging tool.

Katsura et al. used this feature of HLISP for speeding up the handling of recurrence relations in a quantum mechanical calculation [14]. Sasaki implemented an assembly coded package for switching FORTRAN function subprogams into "ON TAB" mode and applied it to speeding up the computation of 6-J symbols [10,20]. His package uses the same data structure as HLISP for tabulation and it modifies the software interface of the specified FORTRAN subprograms transparently to users.

5. Needs for Block-Tabulation

Let case(x) be a function such that its respective values for specific cases $x=X_1$, $x=X_2$, ... $x=X_n$ are Q_1, Q_2, ... Q_n but case(x) = Q_0 for other general or "exceptional" cases of x. This may be programmed as

```
case(x) <= (begin r<=Q0 ; (if x=X1 then r<=Q1 fi); ... ;
            return r as the result end)
```

which makes an O(n) time linear search. Speed up would be effected by specifying "ON TAB case". However, the tabulation would not be effective when "exceptional" cases for x take place for diversified values of x more frequently than the specific cases. A great overhead, both in time and space, would result for tabulating "exceptional" entries. In such a situation, a principle to be called

"block-tabulation" would help. Namely, all the special cases are to be tabulated in "block", or more algorithmically:

(begin (if the block table does not exist then make it fi);

$r \Leftarrow Q_0$;

(if the entry $x=X_i$ is in the block table then $r \Leftarrow Q_i$ fi);

return r as the result end)

There is another benefit: while $O(n^2)$ cost is needed to build up the table for all specific cases in the "TAB ON case" scheme (O(n) time linear search is to be made for each of the n entries independently), this reduces to O(n) time in the "block-tabulation" scheme. If reclaimable tabulation is to be applied, the entire block table must be reclaimed, never by parts.

In the new version of HLISP, the following functions are executed by making use of "block-tabulation".

tabqq(x, $(Q_0\ (X_1, Q_1)\ \ldots\ (X_N, Q_N))$)

is the same as case(x) except in that the tabulation data is given explicitly as an argument.

tabgg(x, $(G_0\ (X_1, G_1)\ \ldots\ (X_N, G_N))$),

to be called "tab go go" is similar to tabqq except in that $G_0, \ldots G_N$ are "go to" labels.

tabmem(x, $(X_1, X_2, \ldots X_N)$)

= tabqq(x, $(F\ (X_1, 'T'), \ldots (X_N, 'T'))$)

is a special instance of tabqq and it checks whether or not x is a member of $(X_1, \ldots X_N)$.

These programming primitives may be somewhat wider in scope than "switch", "case" and "computed go to" compositions in ALGOL 60, PASCAL and FORTRAN. However, they can be regarded as a specific syntactical realization of "decision tables" [11,18].

6. "Unrewritable" Data for Consistent Tabulation

For the consistent use of tabulation the following restriction (R1) and (R2) imposed on the program of the function to be tabulated would be obvious.

(R1) The function is not called for side effects. (R2) The function does not have hidden parameters such as those passed through via global variables.

There is still another restriction (R3) which must be taken in account in case the argument refers to (is a pointer to) a data structure. For the sake of speed and space saving, it would be advantageous to use the pointer to the data structure as the entry of the table. For example, take m(i,j,r). If the N+3 numbers (short integers) i, j, r_0, r_1, ... r_n were used to identify an entry, O(n) time and space would be needed for handling each entry. One would rather use i, j and the pointer r instead. In such cases, however, the data structure (r_0, r_1, ... r_N) pointed at by r should not be rewritten for consistency of tabulation (R3). Note that such a problem does not exist for self-contained and non-referential data such as short integers i, j each being represented as a bit pattern in a single machine word. Also note that even though the program of m(i,j,r) does not rewrite the data structure, programs outside of m(i,j,r) may rewrite it thereby making tabulation inconsistent.

Therefore, in writing tabulative programs in conventional languages, the programmer must be extremely cautious not to violate these restrictions.

It would be highly desirable to implement some automatic means for avoiding inconsistencies upon tabulation. "Unrewritable" data structure, i.e., a data type never to be rewritten after having been created, would provide a partial solution. "Rewrite protection" may be implemented either statically at compile (and linkage edit) time or dynamically at run time. In HLISP, the latter method is used. Namely, the heap for storing data structures (linked lists) are divided into two areas H and L, and the H-type data in the H-area are protected against rewriting at run time. Violation would be regarded as a run time error. It should be noted that in McCarthy's pure LISP [17] i.e., LISP without data structure rewriting primitives "RPLACA" and "RPLACD", all data structures are "unrewritable". Thus, H-type data may be regarded as pure LISP data structures. The function hcopy(L) of HLISP makes an H-type copy of L-type data L. When an H-type datum H, including self-contained non-referential datum such as a short integer, is given to hcopy the result is the argument itself: H = hcopy(H).

When a function m(i,j,r), say is subjected to tabulative computing, all the arguments are H-copied first. In this way, violation of (R3) is checked off automatically. Unique representation of list structures (e.g., only one copy of hcopy(list(1,2,3)) is made in the H-area even if hcopy(list(1,2,3)) is invoked repeatedly.) is the other characteristic feature of H-type data, and there are many applications of the unique representation feature [8,21]. However, for the consistency of tabulation in respect to (R3) the

"unrewritable" feature only is essential.

Rewrite protection in HLISP is rather slow because rewrite violations are checked purely by software at run time. It would be worth noting that contemporary computers are all equipped with memory protection hardware to protect the OS (Operating System) against the illegal memory accesses from the user's programs. Therefore, it should be possible to efficiently implement rewrite protection schemes without modifying the hardware by establishing a suitable protocal between the OS and the user's programs. Besides tabulation, there will be a number of applications of rewrite protection in checking the consistency of programs dynamically at run time. For example, for partial parameterization as implemented in POP-2 [5], the value of "frozen" parameters should be implemented as "unrewritable" data for consistency. Upon defining g(x) to be f(x,y) y is the frozen parameter of partial parameterization. Let y_0 be the value of y when the definition of g(x) was made. Thereafter g(x) means $f(x,y_0)$.

7. Mathematical Tabulative Architecture for Computers

- Another paraphrase for MTAC -

The implementations of the tabulation principles (P1) and (P2) by pure software means described in the previous sections are rather slow. Software hashing, for example, would take 10 memory cycles at the least. The speed can be improved considerably by using hardware hashing like the one disclosed in [7,12]. By making use of parallelism in multi-banked main memory, hashing can be performed as fast as an single indirect addressing through the main memory [7,12].

Further speed up may be possible by using a modified cache

(buffer) memory organization to be called "functional cache". Note that usual cache memories may be regarded as a specific implementation of the tabulation principles (P1) and (P2) with the key being the address of the main memory. The key is to be changed into function invokation in the functional cache. The functional cache may be further backed up with a larger hash table in the main memory to reduce recomputations of functions. Such an organization may be called "buffered hash table" as well. The experiences in software implementations show that rather small table space (e.g., 16K bytes) are adequate in many cases. Thus the cost effectiveness of functional cache may be rather good.

The cache concept has remarkably improved the speed of computations in the past decade. Other tabulative schemes might result in similar improvements as well.

8. Concluding Remarks

- There is nothing new under the sun. -

"On-demand tabulation" must have been practiced since human being ever started to calculate. "Reclaimable and block tabulation" is practiced dayly in making calculations on a black board. Nevertheless, systematizing and restating these old principles are yet believed to be worthwhile. Restated as a "computer programming principles", they were immediately applied by two physists for speeding up their programs [14,20]. Especially, Sasaki clearly demonstrated the general applicability of the principles to different programming language systems.

The idea of using on-demand tabulation for speeding up recursion is not new. It appears in Barron's text book [2] and the idea is

credited to McCarthy. It is given as an example of speeding up the recursion for the partition number q(n,m), i.e., the number of way to express an integer n (n≥1) as a sum of integers no greater than m (m≥1). The recursion reads:

```
q(n,m) <= (if n=1∨m=1 then 1 else
                (if m≥n then q(n,n-1) else
                    q(n-m,m) + q(n,m-1) fi) fi).
```

The tabulation is made explicitly by making use of a linked list instead of an array or hash table. Hence, the time complexity of table retrieval in O(N) rather than O(1), where N is the number of tabulated items.

In the program library of POP-2 there is facility called MEMOFNS (memo-functions) which can specify functions to be tabulated. LRU replacement is made by a linear search through a rote, resulting into O(N) retrieval time instead of O(1) time characteristic to hashing.

As described in section 3, DP (dynamic programming) can be regarded as a specific recursion elimination method which makes use of tabulation in a TS (Topologically Sorted) sequence. A unified view on recursion elimination and DP by making use of R-trees and R-dags would serve to elucidate the underlying programming principles.

Owing to the rapid increase in the cost effectiveness of high speed memories, being realized by advanced LSI technology, tabulative methods are likely to be used increasingly in the future. As exemplified in this paper, more sophisticated computer architecture and programming methology will be needed for the full development of MTAC.

E.G & M.T Sep. 7, 1977

References

1. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

2. Barron, D. W. Recursive Techniques in Programming. Macdonald, London, 1968, pp.17-19.

3. Bird, R. S. Note on Recursion Elimination. Comm. ACM, Vol. 20, No. 6 (1977), pp.434-439.

4. Burstall R. M., Collins, J. S. and Popplestone, R. J. Programming in POP-2. Edinburgh University Press, Edinburgh, 1971.

5. Burstall R. M. and Darlington, J. A Transformation System for Developing Recursive Programs. J. ACM, Vol. 24, No. 1 (1977), pp.44-67.

6. Goto, E. Monocopy and Associative Algorithms in an extended LISP. Information Science Lab. Technical Report 74-03, University of Tokyo (1974).

7. Goto, E., Ida, T. and Gunji, T. Parallel Hashing Algorithms. Information Processing Letters, Vol. 6, No. 1 (1977) pp.8-13.

8. Goto, E. and Kanada, Y. Hashing Lemmas on Time Complexities with Applications to Formula Manipulation. Proc. of ACM SYMSAC 76, Yorktown Heights, N.Y., (1976).

9. Gries, D. Compiler Construction for Digital Computers. John Wiley and Sons, N.Y., 1971.

10. Gunn, J. H. Algorithm 260, 6-J Symbols and 261, 9-J Symbols. Collected Algorithms from CACM, Vol. 2, 1976.

11. Humby, E. Programs from Decision Tables. Macdonald and American Elsevier, London and N.Y., 1973.

12. Ida, T. and Goto, E. Performance of a Parallel Hash Hardware with Key Deletion. Proc. of IFIP Congress 77 (1977).

13. Kanada, Y. Implementation of HLISP and Algebraic Manipulation Language REDUCE-2. Information Science Lab. Tech. Rep. 75-01, University of Tokyo (1975).

14. Katsura, K., Sampei, M., Suzuki, M. and Abe, Y. Application of the HLISP Programming for the Irreducible Representation Matrix of the Permutation Groups and the Partition Function of the Linear Heisenberg Chain (Preprint). Dept. of Applied Physics, Tohoku University (1977).

15. Knuth, D. E. The Art of Computer Programming. Vol. 1, Fundamental Algorithms, 2-nd ed. Addison-Wesley, Reading, Mass., 1973.

16. Knuth, D. E. The Art of Computer Programming, Vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

17. McCarthy, J. et al. LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Mass., 1962.

18. Metzner, J. R. and Barnes, B. H. Decision Table Languages and Systems. Academic Press, N.Y., 1977.

19. Morris, J. H. Jr. and Henderson, P. A Lazy Evaluator.

20. Sasaki, F. "private communicaton". Dept. of Chemistry, Hokkaido University (1977).

21. Sassa, M and Goto, E. A Hashing Method for Fast Set Operations. Information Processing Letters, Vol. 5, No. 2 (1976).

22. Terashima, M. Algorithms Used in an Implementation of HLISP. Information Science Lab. Tech. Rep. 75-03, University of Tokyo (1975).