Acta Informatica 10, 85–94 (1978)

Information Management in Context Trees

Carlo Montangero, Giuliano Pacini, Maria Simi, and Franco Turini
Istituto di Scienze dell' Informazione, Università di Pisa, Corso Italia, 40, I-56100 Pisa, Italy

Summary. Information management in context trees involves three principal problems: retrieval, updating and garbage collection. These problems are discussed in the paper, and solutions are proposed and motivated. A list organization and relative algorithms to implement context trees are presented. Finally, experimental results are reported about the behaviour of a system which exploits context trees.

1. Background

Context is the concept underlying the control structure of most Artificial Intelligence systems and languages [3, 4, 6–10]. Indeed, most of programming systems for Artificial Intelligence [2] are designed to aid the exploration of problem spaces. Whatever implementation is adopted, contexts provide independent environments in which the different possibilities to attain a goal can be explored.

In general, contexts are the base of any system in which there is a hierarchy of environments which envolve independently; for instance, contexts can be used in data base management systems to provide environments in which users of the data bank can perform "hypothetical changes" to test particular developments of the basic model without affecting each other [1].

Context can be implemented by indexing information with context labels. According to [11], data are represented by triples like $\langle K, V, C\rangle$, where K is a key (for instance a variable name), V is the associated value and C is the context in which the pair $K-V$ is asserted.

Context are organized in a tree structure (context tree). Any new context C' is generated as a leaf of the tree; $K-V$ pairs of an ancestor continue to hold in its descendants unless explicitly superseded. In other words, the new context is identical to its father, upon generation.

The value under key K in a context C is retrieved by looking for the triple $\langle K, V, C'\rangle$, where C' is the nearest ancestor of C (possibly C itself), in which a value V has been asserted under K.

0001-5903/78/0010/0085/$02.00

The problem of efficient information management in multienvironment systems based on context trees has been considered in [5, 11]. Triples are grouped by key. This technique seems to be preferable to grouping by context, since the number of triples with key K is small in typical applications. As a matter of fact, on the average it is notably smaller then the number of nodes in the context tree (see Sect. 4).

Both [5] and [11] propose a list organization of triples having the same key K and present retrieval algorithms.

A tree organization for triples and algorithms for information management are described in [12].

This paper discusses some general problems about the design of data updating and garbage collection and shows how they are solved in the implementation of MAGMA-lisp [5].

2. Information Management in Context Trees

Information management in context trees involves three principal operations: data retrieval, data updating and garbage collection.

In the sequel $T(K, C)$ denotes the triple having K and C as its first and third elements respectively; that is if $T(K, C)=\langle K, V, C\rangle$, V is the last value asserted under key K in context C.

2.1. Data Retrieval

According to the definition of context given in Section 1, the data retrieval procedure can be described as follows:

> *Retrieve* (K, C) finds the first node C' in the path from context C to the root of the context tree such that a triple $T(K, C')$ exists, and returns the triple. If $T(K, C')=\langle K, V, C'\rangle$, V is the value associated with key K in context C.

2.2. Data Updating

The simplest technique to update information in context trees would be the following:

a) given a value V to be asserted under the key K in context C, update the triple $T(K, C)$ to $\langle K, V, C\rangle$ if such a triple exists: add the triple $\langle K, V, C\rangle$ otherwise.

However this technique is acceptable only if updating is restricted to the leaves of the context tree. If this technique is used to update information in nonterminal nodes, the propagation in the descendants of C depends on the past updating history. In fact updating propagates to a descendant of C (say C') if *retrieve* (K, C') yields the same triple as *retrieve* (K, C), while C' is not affected at all if *retrieve* (K, C') is not equal to *retrieve* (K, C).

If updating in nonterminal contexts is allowed, it is desirable that the rules of propagation are clearly stated. At least two different rules can be considered:

C-updating (contextual updating), i.e. updating affects the context in which it takes place only;

G-updating (global updating), i.e. updating affects all the descendants of the updated context.

C-update can be implemented by first performing a *retrieve* (K, C) which yields a triple (say $\langle K, V', C\rangle$), and then adding the triple $\langle K, V', C''\rangle$ for any C'' which is a son of C if no triple like $T(K, C'')$ already exists.

G-update (K, V, C) can be performed by deleting all the triples $\langle K, W, C''\rangle$ with C'' a descendant of C.

In both cases updating terminates with operation a).

Obviously *G-update* is more efficient than *C-update*. The updating modality may be left optimal. For instance the default might be *G-update*, *C-update* being performed only when explicitly requested by the program.

2.3. Garbage Collection

The context tree is a dynamic structure, since nodes are added and dropped while executing a program.

Obviously, if the data base is not purged, memory space will become full of triples belonging to deleted contexts. A garbage collector which eliminates triples no longer of interest is needed.

2.3.1. Garbage Collection Generalities

Let $\mathscr{A}$ be the subset of contexts still of interest (alive contexts) when the garbage collection is started.

Given a key K, *collect* (K, A) transforms set $\mathscr{T}_K$ of the triples having key K in a *retrieve equivalent projection onto* A, say $\mathscr{T}'_K$, i.e.

a) any triple $T(K, C)$ with C not belonging to $\mathscr{A}$ occurs no longer in $\mathscr{T}'_K$;

b) $\mathscr{T}'_K$ is *retrieve equivalent* to $\mathscr{T}_K$, that is for any context C belonging to $\mathscr{A}$, *retrieve* (K, C) returns triples having the same value field both from $\mathscr{T}'_K$ and $\mathscr{T}_K$.

Collect (K, A) is intended to drop from the data base the triples relative to contexts which are no longer of interest. It is worth noting, however, that it may add triples to the data base if $\mathscr{A}$ is subjected to no restriction. Let us suppose, for instance, that C does not belong to $\mathscr{A}$ while its sons C' and C'' belong to $\mathscr{A}$. Moreover let $T(K, C) = \langle K, V, C\rangle$ belong to $\mathscr{T}_K$ while there are no triples for C' and C'': the triples $\langle K, V, C'\rangle$ and $\langle K, V, C''\rangle$ must be added while dropping $\langle K, V, C\rangle$ to insure retrieve equivalence. Indeed the value both in C' and C'' was formerly found in the triple being dropped.

Definition. Given any subset $\mathscr{S}$ of contexts, $Sup(\mathscr{S})$ is the root of the *smallest* subtree containing $\mathscr{S}$.

Proposition 1. If $\forall \mathscr{S}_A \subseteq \mathscr{A}: Sup(\mathscr{S}_A) \in \mathscr{A}$ **then**

$\forall \mathscr{T}_K$: there is a retrieve equivalent projection $\mathscr{T}'_K$ such that $\#\mathscr{T}'_K \leqq \#\mathscr{T}_K$.

Proof. Let $\langle K, V, C_z \rangle$ be a triple belonging to $\mathcal{T}_K$ such that C_z does not belong to $\mathcal{A}$ and $\mathcal{Z}$ denote the set of alive descendants of C_z.

If $\mathcal{Z}$ is empty or *retrieve* (K, Sup($\mathcal{Z}$)) does not yield $\langle K, V, C_z \rangle$, delete $\langle K, V, C_z \rangle$.

If *retrieve* (K, Sup($\mathcal{Z}$)) yields $\langle K, V, C_z \rangle$, transform $\langle K, V, C_z \rangle$ into $\langle K, V, \text{Sup}(\mathcal{Z}) \rangle$.

The set $\mathcal{T}_K''$ obtained this way is retrieve equivalent to $\mathcal{T}_K$ and has no more elements than it. The desired retrieve equivalent projection $\mathcal{T}_K'$ is constructed iterating this procedure.

The above discussion shows that the complexity of garbage collection depends on the structure of subset $\mathcal{A}$, i.e. on the policy adopted to delete contexts.

The following subsection discusses the deletion policy of MAGMA-lisp.

2.3.2. Context Deletion Policy

In general, context organized systems will provide possibilities to add and delete contexts. With regard to deletion, it will be convenient to choose techniques consistent with the condition of Proposition 1. The following deletion policy is not too restrictive and is consistent with Proposition 1:

> it is possible to replace a subtree $\mathcal{S}_T$ by another subtree $\mathcal{S}_T'$ (possibly empty) such that the root of $\mathcal{S}_T'$ is an element of $\mathcal{S}_T$.

The above proposed technique allows pruning the context tree both towards the leaves and towards the root. Both possibilities are useful in practice. For example, consider nondeterministic applications: on failures terminal subtrees rooted in failing contexts must be deleted; on successes, antecedents of the successful context must be eliminated in order to disregard further possibilities to attain the goal.

2.3.3. Garbage Collection Triggering

After a deletion some triples belonging to deleted contexts survive in the system. It is interesting to note, however, that such triples do not prejudice the future correct behaviour of the retrieve procedure, so that it is not necessary that a garbage collection immediately follows each deletion. As is intuitive, purging the data base may improve performance, but, on the other hand, care must be used to insure that the cost of garbage collection does not prevail over the obtained gain. So, garbage collection delaying must be tuned in order to find a good compromise between the cost of garbage collection and the fall in performance due to its delay. For instance, garbage collection might be triggered when a predetermined number of contexts have been deleted.

3. The Proposed Implementation

All the algorithms described in this section have been exploited in implementing the MAGMA-lisp system [5].

The algorithms to retrieve and update information can be easily justified. On the contrary, proving the correctness of the algorithm for garbage collection requires some work. The proof is not given here.

Two arrays P and D are maintained, such that: $P(C)$ is the number of nodes that precede the node C in the preorder traversal of the context tree;

$$D(C)=\begin{cases}P(C), & \text{if } C \text{ is a terminal node;}\\ \max\{P(C_j)\mid C_j \text{ is a descendant of } C\} & \text{otherwise.}\end{cases}$$

Property 1. For any contexts C_i and C_j, C_j is a descendant of C_i iff:

$$D(C_i)\geqq P(C_j)>P(C_i).$$

Triples are grouped by key. The triple $T(K, C_0)=\langle K, V_0, C_0\rangle$, where C_0 is the root of the context tree, is always present in the set $\mathscr{T}_K$ of the triples having key K; V_0 may possibly be the "undefined value".

Each $\mathscr{T}_K$ is ordered by *decreasing* values of the associated elements of P.

3.1.1. Retrieve (K, C): Basic Algorithm

Step 1.
skip the triples of $\mathscr{T}_K$ **until** a triple $T(K, C_j)$ is found with $P(C)\geqq P(C_j)$

Step 2.
if $C_j=C$
then return the triple
else return the first triple (say $\langle K, V, C_k\rangle$) such that

$$D(C_k)\geqq P(C).$$

This algorithm is $\mathbf{O}(\#\mathscr{T}_K)$, since at most all the triples in $\mathscr{T}_K$ are considered once.

3.1.2. Retrieve (K, C): Improved Algorithm

The performance of the previous algorithm can be improved exploiting a technique inspired by [12].

Two more pieces of information (named L and τ) are associated to any $\mathscr{T}_K$. During *retrieve* (K, C')

1) L is set to $P(C')$,

2) τ is set to point to the first triple of $\mathscr{T}_K$, say $T(K, C'')$, with $P(C'')\leqq L$.

To perform *retrieve* (K, C):

if $P(C)>L$ **then** *retrieve* starts from the first element of $\mathscr{T}_K$:

if $P(C)\leqq L$ **then** *retrieve* starts from the element pointed by τ.

This simple improvement allows to avoid the scanning of part of $\mathscr{T}_K$ in many cases. Experience shows that a remarkable gain in the performance is obtained (see Sect. 4).

3.2.1. G-Update (K, W, C)

Step 1. **For** each triple $T(K, C_j)$ with $P(C_j)>P(C)$
if $D(C)\geqq P(C_j)>P(C)$ (i.e. C_j is a descendant of C by property 1).
then drop $T(K, C_j)$
else skip the triple;

Step 2. Let $T(K, C_j) = \langle K, V, C_j \rangle$ be the first triple such that $P(C) \geqq P(C_j)$
if $C_j = C$
then change $\langle K, V, C_j \rangle$ into $\langle K, W, C_j \rangle$
else insert $\langle K, W, C \rangle$ just before $T(K, C_j)$.

Again, this requires at most a single scan of $\mathscr{T}_K$, so that the algorithm is $\mathbf{O}(\#\mathscr{T}_K)$.

3.2.2. *C*-Update (*K*, *W*, *C*)

The set S of the sons of C, ordered by decreasing values of the associated elements of array P, is needed; it is obtained from the context tree.

The first step of the algorithm inserts a triple for each son of C not yet in $\mathscr{T}_K$. The value field is used to link them in a list L.

In *step 2*, once the old value V under K in context C is retrieved, L is scanned back to set the value field of each triple to V. Finally the new value is stored.

L is initialized to $\emptyset$.

Step 1. **For** each $C_n \in \mathscr{S}$, let $T(K, C_j)$ be the first triple with $P(C_n) \geqq P(C_j)$ **do**
if $C_n \neq C_j$
then insert $\langle K, L, C_n \rangle$ just before $T(K, C_j)$, updating L;
else skip the triple;

Step 2. Let $T(K, C_j) = \langle K, V, C_j \rangle$ be the first triple such that $P(C) \geqq P(C_j)$ **do**
if $C_j = C$
then set the value field of the triples in L to V and change $T(K, C_j)$ into $\langle K, W, C_j \rangle$
else insert $\langle K, W, C \rangle$ just before $\langle K, V, C_j \rangle$, *retrieve* the previous value of K in C (by step 2 of Sect. 3.1) and modify the triples in L accordingly.

Since each iteration of step 1 can restart from the last considered triple in T_K, because of the ordering of both $\mathscr{T}_K$ and $\mathscr{S}$, the algorithm is $\mathbf{O}(\#\mathscr{T}_K + \#\mathscr{S})$.

3.3. *Collect* (*K*, $\mathscr{A}$)

The set $\mathscr{A}$ of alive contexts satisfies the condition of Proposition 1, and is ordered by decreasing values of the associated elements of P, as usual.

The algorithm scans $\mathscr{T}_K$ and $\mathscr{A}$ in parallel. At each iteration three cases are distinguished (see Algorithm). $\mathscr{T}_K$ is modified in case c only. Thus, in order to show the equivalence between the algorithm and the procedure outlined in Proposition 1, it is enough to prove that:

a triple $T(K, t) \in \mathscr{T}_K$ is considered in case c if and only if $t \notin \mathscr{A}$;

the modification of $\mathscr{T}_K$ (i.e. updating or deleting of $T(K, t)$ is performed according to Proposition 1.

The detailed proof is not given here. Anyway, the principal point is that LA can be used to decide whether $T(K, t)$ has to be updated or deleted. Moreover, if $T(K, t)$ must be updated, then LA is the *Sup* of the alive descendants of t.

Algorithm. α and τ are initialized to the first element in $\mathscr{A}$ and to the context component of the first triple in $\mathscr{T}_K$ respectively. LA is initialized to $\emptyset$.

Case a. **If** $P(\alpha) > P(\tau)$
then set LA to α and repeat after updating α to the next element of $\mathscr{A}$;

Case b. **If** $P(\alpha) = P(\tau)$
then set LA to $\emptyset$ and repeat after updating α to the next element in $\mathscr{A}$ and τ to the context component in the next triple of $\mathscr{T}_K$;

Case c. **If** $P(\tau) > P(\alpha)$ or $\alpha = \emptyset$ (i.e. $\mathscr{A}$ is completely scanned)
then if $LA \neq \emptyset$ and LA is a descendant of τ
then modify $T(K, \tau)$ to $T(K, LA)$ and reset LA to $\emptyset$
else drop the triple $T(K, \tau)$ leaving LA unchanged;
repeat after updating τ to the context component in the next triple of $\mathscr{T}_K$.

The algorithm stops when all the triples in $\mathscr{T}_K$ have been considered.
Since both $\mathscr{T}_K$ and $\mathscr{A}$ are scanned once in parallel, the algorithm is

$\mathbf{O}(\#\mathscr{T}_K + \#\mathscr{A})$.

3.4. Maintenance of Arrays P and D

Arrays P and D change only when nodes are added or deleted from the context tree. To add a new leaf C' as the last son of C, no traversal of the context tree is needed, since updating can be performed in a single scan of the arrays. according to the following algorithm.

Step 1. **For** any j such that $P(C_j) > D(C)$ (i.e. all the nodes encountered after the last descendant of C in the traversal) **do**:

$$P(C_j) := P(C_j) + 1, \quad D(C_j) := D(C_j) + 1;$$

Step 2. **For** any j such that $D(C_j) \geqq P(C) \geqq P(C_j)$ (i.e. all the nodes on the path from the root to C inclusive) **do**:

$$D(C_j) := D(C_j) + 1;$$

Step 3. $P(C')$, $D(C') := D(C) + 1$.

To remove nodes from the context tree, a complete rearrangement of P and D is needed by traversing the context tree.

The addition of a new node takes a time proportional to the context tree size. Nevertheless, creating a new context seems to produce minor overhead in overall execution time in practical cases. Finally context switching takes negligible time, since it reduces to updating the active context indicator.

4. Some Experimental Results

We report here some experimental data on the behaviour of MAGMA-Lisp with respect to contextual information management.

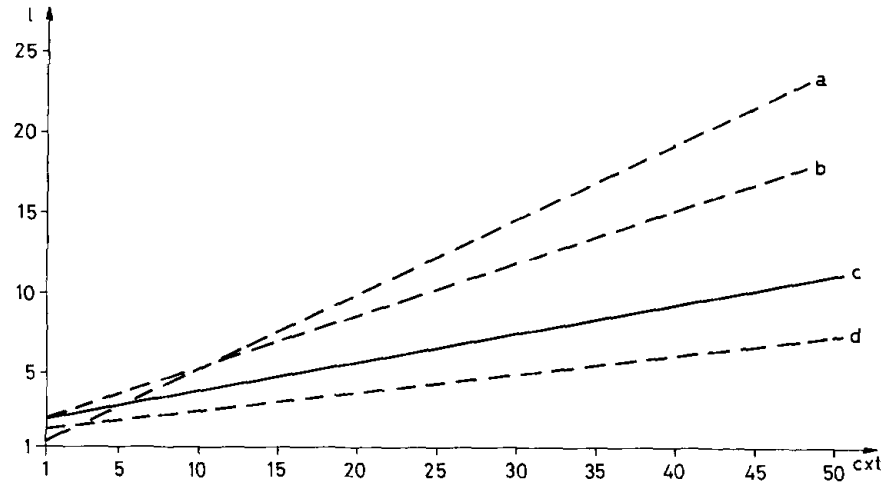

Fig. 1. *a* $S_{l.cxt}=0.20$; *b* $S_{l.cxt}=0.96$; *c* $S_{l.cxt}=3.41$; *d* $S_{l.cxt}=0.46$

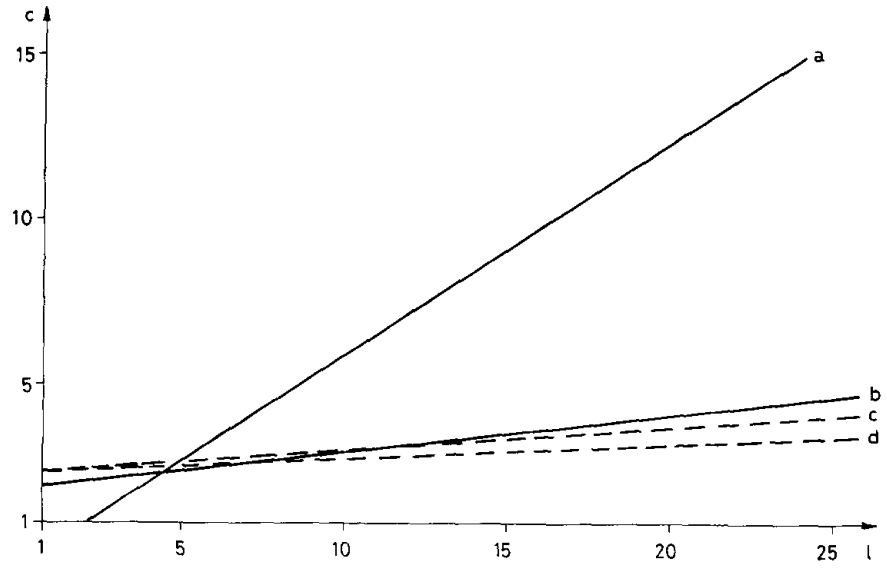

Fig. 2. *a* basic algorithm with breadth first search; *b* improved algorithm with breadth first search; *c* basic algorithm with depth first search; *d* improved algorithm with depth first search

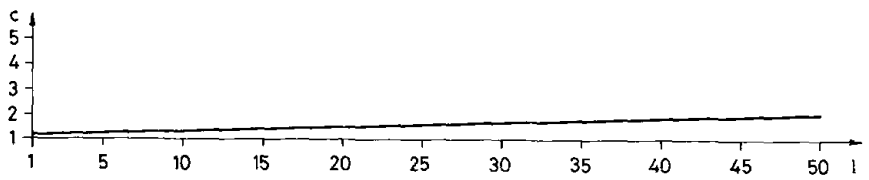

Fig. 3

We considered a sample of some hundred runs of typical problem solving programs including, for instance, the solution of the n queens problem, simple string matching, parsing with Transition Network Grammars and maze searching.

Data have been gathered in the following way: at each access in a list of triples $\mathscr{T}_K$, the number of nodes in the context tree, the number of triples in the list and the cost of the retrieval (i.e. the number of inspected triples) are recorded to compute their average values (cxt, l, c) over the relative run.

The experimental results provide general information about the behaviour of context based systems (Fig. 1) on one hand, and specific estimations of the performance of the implementation described here (Figs. 2 and 3) on the other.

Figure 1 refers to the dependency of the length of the $\mathscr{T}_K$'s versus the size of the context tree. The three dotted lines are obtained by fitting the points resulting from several runs of three of the experimented programs, increasing each time the size of the computation by suitable choices of the input data. The standard error ($S_{l.cxt}$) shows that in most cases the experimental points due to a single program are well fitted by a straight line. Moreover, it turns out that the slope of the line undergoes large variations from one program to another. The solid line represents the best linear fitting of the whole set of experimental points. The standard error is relevant here, due to the mentioned large variations of the slope, passing from one program to another.

Figure 2 shows the dependency of the cost c versus the length of $\mathscr{T}_K$'s. It compares the performances of the basic retrieval algorithm and the improved one. Data are grouped in two classes, according to the strategy used in the corresponding programs: dotted lines refer to depth first search, solid lines to breadth first search. The improved algorithm gives little gain over the noticeably good performance of the basic algorithm in the case of depth first search. Indeed, as may be easily understood, it almost always happens with backtracking that *retrieve* returns one of the first triples of $\mathscr{T}_K$. On the other hand, the gain is relevant in the case of breadth first search, because deep accesses to the $\mathscr{T}_K$'s are frequent in this case.

Finally, the overall average cost with the improved retrieval method versus the size of the context tree is given in Figure 3. It turns out that the average cost is low, even for a reasonably large number of contexts. This agrees with another interesting experimental result: on the average and with a very small variance the right value is found by a single test in roughly 75% of the cases.

5. Conclusions

The paper discusses problems pertaining to information management in context trees. The proposed implementation is a general solution to these problems.

Algorithms have a linear complexity and the experimental results show that they are a reasonable trade-off between efficiency on one hand and memory occupation and simplicity of organization on the other.

References

1. Abrial, J.R.: Data semantics. In: Data base management (J.W. Kimbie, K.L. Koffeman, eds.). Amsterdam: North Holland 1974
2. Bobrow, D.G., Raphael, B.: New programming languages for Artificial Intelligence Research. Comput. Surveys **6**, 153–174 (1974)
3. Davies, D., Julian, M.: Popler 1.5 Reference Manual. Univ. of Edinburgh, TPU Report 1, May 1973
4. Hewitt, C.: Procedural embedding of knowledge in PLANNER. Proc. 2nd Int. J. Conf. on Artificial Intelligence, London, pp. 167–182, 1971
5. Montangero, C., Pacini, G., Turini, F.: MAGMA-Lisp: a machine language for Artificial Intelligence. Proc. 4th Int. J. Conf. on Artificial Intelligence, Tbilisi, pp. 556–561, 1975
6. Montangero, C., Pacini, G., Turini, F.: Two-level control structure for nondeterministic programming. Comm. ACM **20**, 725–730 (1977)
7. Reboh, R., Sacerdoti, E.: A preliminary QLISP manual. Stanford Research Institute, Artificial Intelligence Center, Tech. Note 81, August 1973
8. Rulifson, J.F., Waldinger, R.J., Derksen, J.A.: QA4: a procedural calculus for intuitive reasoning. Stanford Research Institute, Artificial Intelligence Center, Tech. Note 73, November 1973
9. Smith, D.C., Enea, H.J.: Backtracking in MLISP2. Proc. 3rd Int. J. Conf. on Artificial Intelligence, Stanford, pp. 671–685, 1973
10. Sussman, G.J., McDermott, D.V.: From PLANNER to CONNIVER, a genetic approach. Proc. AFIPS FJCC 72, Vol. 41, Pt. II, AFIPS Press, Montvale, N.J., pp. 1171–1179, 1972
11. Wegbreit, B.: Retrieval from context trees. Information Processing Lett. **3**, 119–120 (1975)
12. Wegbreit, B.: Faster retrieval from context trees. Comm. ACM **19**, 526–529 (1976)

Received July 27, 1976