PLANNER-BESM-6 SYSTEM: IMPLEMENTATION METHODS

Vladimir N. Pilehikov

MoscowUniversity
Department of Computation Mathematics and Cybernetics
Moscow, USSR

Abstract

This paper describee some new methods used in PLANNER-BESM-6 system for the implementation of PLANNER. Backtracking is implemented with a single stack and no copying of activation frames is needed. Lists are represented as arrays of pointers; the garbage collector for this representation requires no additional memory. Restrictions on variable values are accomplished by using 'semi-defined' structures. These methods have enabled to achieve the high efficiency of the system work.

1• Introduction

Newer programming languages [1,2] intended for use in Artificial Intelligence research have introduced many new facilities that make much more easy the construction of sophisticated AI systems. Therefore practical implementation of these languages and creating efficient methods for this are the necessary and important tasks for AI progress.

Among the new languages PLANNER [3] has gained the widest popularity. PLANNER was the firBt language introducing a majority of the new concepts and methods. Now PLANNER attracts much attention: many papers propose methods of its translation, there are some practical implementations of it and others are being created.

PLANNER-BESM-6 system is an interpreter. In some important aspects this system differs from other PLANNER systems. The system has been designed to be an efficient practical tool, so the efficiency of the language implementation, which is the heel of Achilles for PLANNER, has been given more attention rather than aiming to implement all without exception features of PLANNER. Some features which are not the main within the language but require the superfluous memory and time for their implementation have not been introduced into the input language of this system. For example the input language uses only recursive and backtrack regimes and prohibits actors' utilization in matching patterns. This has enabled to use methods that increase the efficiency of the system work.

Together with the well-known techniques (coordinates in data base organization [3,4], context number for fail-points [5] and so on) PLANNER-BESM-6 system uses some new methods. This paper briefly describes three of such methods. Backtracking is implemented with a single stack and without copying information in the stack. Lists are represented as arrays of pointers which contain the length of lists; the garbage collection used for this representation requires no additional space. Restrictions on variable values are accomplished with structures that are not fully defined.

2. Implementation of backtracking

There are some schemes for implementing backtrack control. For example the paper [5] proposes a scheme with two stacks. But this scheme requires multiple transfers of information from one stack to another. The scheme of [6} uses one stack and is intended for implementation of various control regimes. This general scheme has been used to implement language POPLER 1.5 [7] which has sophisticated control structure. However, being adapted only to backtracking, this scheme spends the superfluous space and time because it requires multiple copies of activation frames.

PLANNER-BESM-6 system uses also one stack for keeping frames of activations but no copying is needed, so this scheme requires less memory and time than the above schemes. The scheme is as followes [81.

There is a list called 'archive' that roughly corresponds to 'failist' of scheme [6] . Archive saves information on changes of variable binding, data base and so on, made by the program. For each changed object archive keeps its location in memory and its previous state. Using this the system will restore the previous states of the objects when a failure occurs within the program.

Stack contains information needed for functions' elaboration. Upon entry to a function certain storage, called a frame of the function activation, is allocated in the end of stack. Any frame occupies

always one continuous piece of stack space. The head of a frame contains activation name, a pointer Ar to archive and links of this frame: a pointer I (see below), a pointer AL which specifies the variable bindings accessible within this frame, a pointer CL to the calling frame, and return address Rt to the caller (thus a continuation point is saved in the called frame). A frame may also contain information on exit actions of the activation. Another space of a frame holds temporary intermediate results and local variables' bindings, if any, of the function. During the whole activation the size of its frame is constant (while the contents of frame may vary), so the size is fixed on activation entry and then does not vary.

If a function hasn't set any failpoint during its elaboration and if the function exits, then the frame of this function is removed out of stack, i.e. a pointer s to the end of stack will again point to a cell upper this frame. If during an activation a failpoint has been set, then the activation frame is retained in stack. In this case the pointer s is not changed. A consequence of this is that the frame of active function is not always located in the end of stack. For this reason there is another pointer r which always points to the *frame* of function that is active now. The active function uses this pointer in order to access its own frame. When a new frame is created, the value of r becomes a link CL, and then r begins to point to the head of the new frame; later, when the function exits, CL will be transfered to r. Therefore without failpoints the pointers s and r are changed synchronously, which corresponds to normal recursive regime.

When a function has to set a failpoint, it allocated information on this failpoint in the last cell of its frame and then fixes a pointer f to this cell (the third stack pointer - f - always points to a cell of the last existing failpoint). This information consists of address PP of the previous failpoint cell and 'reaction address' RA that is an address of the interpreter instractions which will perform certain actions when a failure returns control to this failpoint. Pig. 1 shows the contents of the frame for [AMONG (4 IJCAI (USSR TBILISI) 1975)] in the moment when this function has set a failpoint before returning the first selected value - 4.

It is possible by comparing the pointers r and f to determine whether or not a function has set a failpoint and hence to determine whether or not a frame of the function should be retained in stack. If on a function exit $r < f$ then the function has a failpoint so the pointer s is not changed, otherwise s varies. Pig. 2 shows the states of stack and pointers e, r and f at different moments of the elaboration of a function P which calls functions G, H and I in turn, the function G setting a failpoint and I calling a function J which generates a failure: a) G exits; b) H exits; c) J generates a failure; d) control is returned to G.

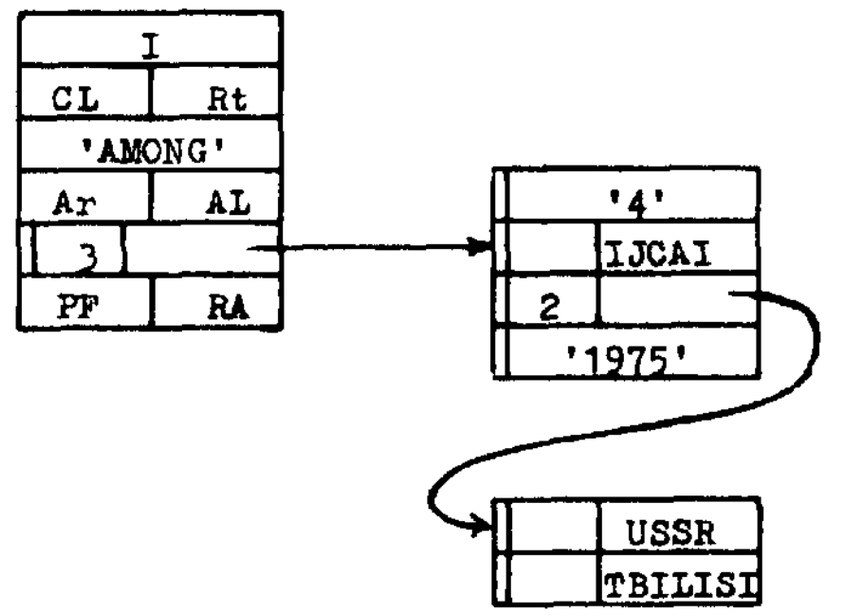

Figure 1. Examples of frame and list representation.

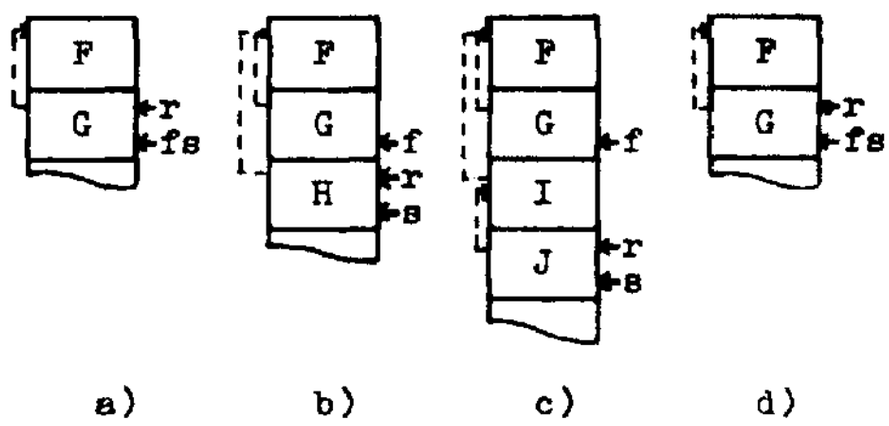

Figure 2. States of stack and pointers s, r and f.

When a failure occurs, the interpreter cleans stack up to the frame which keeps the last failDoint (the value of f is transfered to s) and then returns control to the reaction on failure within this failpoint. Since information on the failpoint is located in a frame of activation that has set the failpoint, the reaction gets an access to all information of this frame, in particular to information needed for redoing the elaboration of the function. Using this information the reaction restores the pointer r_t restores access environment with link CL, and then with scanning archive it restores previous values of variables, previous state of data base and so on. Due to these actions the state of execution is restored. The further actions of reaction are different for different functions. For example, function AMONG will chop the next element off the list stored in its frame and then will return this

element as new result. But if the list is empty then the function will destroy its failpoint (address PF will be tranefered to pointer f) and failure will propagate upper.

An addition is necessary to the described scheme. Let us consider the elaboration of [AND e1 e2 e3 e4] . This function calls its arguments in turn until false occurs- AHD sets no failpoint so it cannot catch a failure on its own. However its arguments may set failpoints and generate a failure. Hence AND cannot manage transfers of control among its arguments. For example, let e1 sets a failpoint and e3 generates a failure that returns control to e1. If the repeated computation of e1 succeeds then control is returned to AND again, but this function will not know that it should call e2 and not e4.

This mistake is easily rectified. When a function calls another function, the calling one gives some information (a pointer I) to the called one to keep it. When the lower function finishes it returns this pointer together with its result. As a rule, I is a pointer to the list of arguments followed the argument evaluated now. The returned pointer informs the calling function which argument should be evaluated at the next step. Hence the correct work of the calling function is not broken by transfering control from one argument to another.

All functions of PLANNER for backtrack regime, in particular non-local goto and failures directed to a point, were implemented within the bounds of the described scheme.

3- List representation

In PLANNER the main operations on lists are more complex than, for example, those in LISP, There are scanning lists from both the ends, selecting any element or sequent from list, matching list-patterns. The usual list representation (as in LISP systems) is not effective for elaborating these operations. For example a match of lists requires easy determining the list lengths but it is impossible with this representation. On the other hand, the operation cons, efficiently implemented with this representation, is not typical for PLANNER. Moreover the usual list representation leads to scattering list cells all over the memory, which is inconvinient for the paging of virtual memory used in PLANNER-BESM-6 system.

In view of this, PLANNER-BESM-6 system uses a different list representation, namely, a list is represented as an array of pointers (see Fig. 1). A pointer to a list consists of three parte: the initial address of array, the length of array and a type indicator (each data type has its own indicator). A list length used tn a pointer makes easy to match patterns and to scan lists from the end. Successive location of list elements makes easy to select any element or sefment. Due to such representation lists and tuples are not distinguished. In addition, this representation partially (on the highest level) localizes list cells in memory.

A pointer to integer is the same integer with fixed exponent which is also an indicator of type 'integer'. Pointers to other data types consist of two parts, viz., type indicators and references to property cells of data. Free space of these pointers is used for various aims, for example to list all labels of function PROG.

The list representation described requires the free space of list memory to be one continuous sequent. The interpreter fills this space from bottom to top, hence the garbage collector must pack relevant pointers and move them down, modifying references of these pointers. Multiple references to inside of lists make difficulties for this moving and for reference changing, so the garbage collection used in FLANNER-BESM-6 system takes three stages. But this garbage collection is simple and doesn't require additional space, and this differs it from other methods of garbage collection with packing.

The first stage is to mark cells of lists needed for the further runnig of program. This is performed as in IaSP systems, besides the total amount of marked cells is counted.

The second stage is linear scanning list space from top to bottom and changing all references down (i.e. references to cells with larger addresses). This changing is performed as follows.

Let a current examined cell a is marked and also contains a reference to cell b (b>a ; in this stage all references up are ighored). Then the garbage collector interchanges both the references of cell a and b, and if this reference to cell b is the first one from top then both the cells a and b is marked by '+' (this marker is distinct from the marker used in the first stage). Then cell a+1 will be examined.

If a current examined cell b contains the marker '+' then at this moment the cell b contains also the initial address of list of cells a_1 a_2, ... , a_n (a_i<$a_{.+1}$, a < b) which have had the references to cell b before. At this moment address b' which should be assigned to pointer of cell b is known: b' is defined by the total amount and the amount of already examained marked cells. So the garbage collector in scanning the chain of cells a , ... , a„, a- puts the address b^1 into them. The last cell $\&_1$ of this chain is marked and contains the re-

ference that has before been placed in cell b_t so this reference ie transfered to cell b, marker • + • is removed out of cells a_1 and b, and then the contents of cell b is treated on common base.

In PLANNER-BESM-6 system the list space is placed in the bottom of memory, hence all references from other spaces to this space are references down. In order to avoid scanning other spaces in the second stage, all references of those spaces to the list space are treated in the first stage as described above.

The third stage of the garbage collection is linear scanning the list space from bottom to top, moving all marked pointers (to the space bottom) and changing all references up. This stage is similar to the previous stage but each reference up is moved to new location first and then its new address is used.

4. Restrictions on variables

Matching two patterns is used in function MATCH or during theorems' invocation. One of problems here is the implementation of restrictions on variable values. When a pattern matches another pattern some variables get no values but their future values are constrained. For example on elaboration of [MATCH *X *Y] (prefix '*' means 'to assign value to variable') no variable gets a value but their future values will be equal. Another type of restrictions appears in matching a variable with a list which contains variables. For example on elaboration of [MATCH * I (*Y A .Y)] the future value of X will be a list with three elements, the second of which is atom A, two others are equal and are the future value of Y (prefix '.' means 'to get value of variable').

Since the input language of PLANNER-BESM-6 eystem prohibits actors' utilization in matching two patterns, restrictions on variable values may be of the above types only. This restrictions are implemented in such manner.

New data type 'semi-defined structure' is introduced. This is a structure a part of which is not defined, namely, a inner (not accessible to users) variable without a value or a list some elements of which are semi-defined structures. If a variable gets no a fully defined ('real') value in matching then ite value (SD-value) will be a semi-defined structure. This SD- value is constructed with the pattern which has been confronted to the variable: all variables of the pattern are substituted by their values, in particular by SD-values, wherever possible; the rest of variables get references to some inner unassigned variables as values, and then these references are inserted into the pattern. Thus any variable has a value always but SD-value is not accessible to users.

In matching SD-value behaves as well as 'real' value: object confronted to a variable with SD-value must match SD-value. A consequence of this is that a variable may get only the 'real' value that matches the existing SD-value. In general case, any new restriction on variable value is immediately checked on compatibility with the existing SD-value, and if they do not conflict then their 'intersection' will be the new SD-value of the variable. This check often allows to define the full value of variable in proper time* For example this takes place for [MATCH (*X .X) ((*YA) (B *Z))} .

Cross-references among 'real' variables are accomplished by SD-values since in general case each inner variable is referenced by some semi-defined structures. If a semi-defined structure is filled in fully or partially then all or some inner variables referenced by this structure get values, and hence some other semi-defined structures are also filled in fully or partially. Since cross-references among 'real' variables are accomplished indirectly, through inner variables the existence of which doesn't depend on the existence of 'real' variables (but depends on amount of references to them), so it is not necessary to retain 'real' variables in memory only because of the program has defined links among other 'real' variable by those. So when a theorem has set no failpoints then its frame ie removed out of stack even if there are cross-references among global variables through its local variables, because the existing inner variables are holding this links.

References

1. Bobrow D., Raphael B. New Programming Languages for AI Research. ACM Computing; Surveys, v.6, No.3, 1974
2. Fabrin v., Serebriakov V., Yufa V. LORD - AI Programming System. VII Symposium on Cybernetics, Tbilisi, USSR, 1974
3. Hewitt C. Description and Theoretical Analysis (using schemata) of PLANNER. MIT AI Lab., Cambridge, Mass., 1972
4. Baumgart B. MICRO-PLANNER Alternate Reference Manual. Stanford AI Lab*, Stanford, Calif., 1972
5. Smith D., Enea H. Backtracking in MLISP2. Proc. IJCAI-73. Stanford
6. Bobrow D., Wegbreit B. A Model and Stack Implementation of Multiple Environments. CACM. v.16, No.10, 1973
7. Davies D. POPLER 1.5 Reference Manual. University of Edinburg, Edinburg, Scotland, 1973*
8. Pilshikov V. Backtracking in PLANNER-BESM-6 System, in Symbol Information Processing, v.2. Computing Center, Ad&demy 6f Sciences, Moscow, 1975