APPLICATION OF THE PL-I LANGUAGE TO THE IMPLEMENTATION OF STRUCTURED-MEMORY ALGORITHMS

V. V. Bublik and S. S. Gorokhovskii

The article discusses problems associated with the selection of programming methods and language facilities for the development and implementation of software systems. The results are based on the experience gained in the design and application of the PROEKT system [1] software in the course of whose development a new method has been formulated for the design of discrete systems, including program software; the new method is called the method of formalized technical tasks [2] and encompasses a group of concepts and facilities forming the base of instrumental programming languages. Already in the early versions of PROEKT, system implementation languages were developed in parallel with the design of the system proper on the basis of an expansible programming system [3]. This system effectively realized special-purpose languages for processing portions of machine words, texts, tables, trees, etc., which together formed the high-level machine-oriented programming language L1 [1, Chap. 3] which meets the demands of system programming.

The conditions of realization of the PROEKT system in the Unified Computer System (UCS) were fundamentally different. In the conditions of the multilanguage environment of the UCS operating system (UCS OS), of principal interest was the utilization of regular programming systems for high-level languages. As to the set of available facilities, the above demands are most fully met by the USC OS PL-I language many of whose mastering difficulties are alleviated by the excellent guidebook [4].

The most useful for system programming are PL-I facilities which are most efficiently implemented, especially so if one considers that besides obviously inefficient constructions implemented by the PL-I subroutine library (processing of nonaligned bit-lines, definition by similitude, variable-length lines, etc.), there are constructions whose modification, while insignificant from the point of view of the final result, can cause a significant change in the efficiency of working programs (multiplication of objective code and/or a call to the PL-I subroutine library instead of direct generation of objective code). Certain general comments on the improvement of objective program efficiency can be found in [4] and other manuals.

Let us consider the choice of data structures and language constructs of PL-I as applied to the implementation of algorithms on structured memory. Let us consider a class of data structures that are generally most difficult to process. We speak about nonuniform data structures whose format changes in the course of processing. Such data structures can be mathematically defined as a labelled graph or a multigraph of a certain kind, for example, a composite object [5], a data structure arranged on a data graph [6, 7], extended directed graphs [10], etc. Such data structures can usually be defined by specifying certain data elements (primary) and the rules of composition of elements into aggregates or, in other words, the operations of data structure generation. The resulting structure can be thought of as a certain totality of nodes containing marks (operational or primary) and directed links between nodes. Nonuniform data structures, henceforth called composite objects, have one important peculiarity resulting from the nature of their processing. The following principal operations are usually executed on composite objects: addition of a node (link) to the composite object, removal of a node (link), replacement of a node (link) with another node (link), movement along nodes (links) whose superpositions allows arbitrary transformation of composite objects, in particular, the generation or annihilation of a composite object or its sections, the change of its size, etc.

Facilities for processing data structures of the composite object kind are available in LISP, PL-I, ALGOL-68, and many other languages. More advanced facilities (high-level data language, bypass and relation operators) are available in programming languages L1 (TREE)
languages developed with the participation of the authors. The results of the investigation of PL-1 and of its implementation described here were utilized, in particular, in the development of the L2B language programming system.

The information environment contains:
1) facilities for naming composite objects implemented by means of pointers to the respective starting nodes;
2) mechanisms for placing in memory the nodes of composite objects: their marks (including primary) and directional links;
3) facilities for moving along links between the nodes of composite objects;
4) facilities for splitting composite objects into parts which are coordinated with exchange operations between the main and external memories.

Composite objects are stored in memory in the form of lists, i.e., to each node corresponds a certain memory region which holds the node label and a field of links (references of subordinate nodes).

Block Memory

A standard method of list representation in PL-1 is block memory. Nodes are represented by block structures composed of problem data and PL-1 pointers. The technique of operating with block structures and PL-1 pointers (as a means of list processing) has already been discussed in literature (see, e.g., [4]). The PL-1 block memory facilities allow:
1) direct creation and annihilation of composite object nodes by direct application of PL-1 operators;
2) simple object code for accessing structures in correspondence with pointers;
3) relative addressing in PL-1 areas (displacements).

In experimental processing of composite information structures using the facilities of PL-1 block memory we have revealed the following deficiencies:

a) inefficient memory utilization in view of the fact that the size of a reference (pointer or displacement) is independent of the size of the memory area actually being addressed;

b) the need of constant explicit conversion of displacements into pointers and back. The point is that a structure located in some region is identified by a displacement associated with this region. Unfortunately, displacements can be used neither in placement and clear operators nor in qualified references to block variables, i.e., in expressions of the form $P + A$, where $P$ is a pointer and $A$ is block variable;

c) inefficient PL-1 memory placement and clear operators. These operators are implemented by object code (30-40 bytes) for calling a library subroutine of 1024 bytes. The placement operator actually implements the least successful unconditional memory request procedure which can lead to uncontrollable consequences of memory shortage;

d) garbage collection difficulties. One and the same variable can be located, generally speaking, in many places but one must take care of preserving the pointers to previous locations since otherwise they will turn into garbage inaccessible to PL-1 programs. This is due to the lack of flexibility in inspecting all locations existing in the memory. They can be bypassed by moving along references only, but it is impossible to inspect the entire memory in search for a certain node or for garbage collection;

e) limitations imposed by the UCS OS. Memory for data placement is allocated in batches whose size is a multiple of eight bytes irrespective of how much memory is actually needed.

Structure Arrays

As an alternative to automatic PL-1 language solutions let us consider certain constructs for representing composite objects defined by the programmer. If the nodes of composite objects are placed in arrays the access to each such node can be implemented with the aid of an indexed expression. The efficiency of locating composite objects in arrays depends on the array size and on the structure of array elements.