PROGRAMMING LANGUAGE PROGRAPH: YET ANOTHER APPLICATION OF GRAPHICS

Tomasz Pietrzykowski
Acadia University
Wolfville, Nova Scotia

Stanislaw Matwin
University of Ottawa
Ottawa, Ontario

Tomasz Muldner
Acadia University
Wolfville, Nova Scotia

ABSTRACT

The paper describes an innovative programming language PROGRAPH. The design of PROGRAPH is based on data flow concepts and is meant to be a software tool for users of data flow computers. Inherent parallelism of programs, designed for such an environment, is enhanced by graphics, used to specify PROGRAPH programs. Several examples of PROGRAPH programs are presented. Briefly discussed is the graphics editor. This editor is a user's tool to define programs in PROGRAPH.

RESUME

L'article decrit PROGRAPH - nouveau langage de programmation. Le design de PROGRAPH est fonde sur des concepts de flux de donnees et se veut un outil de programmation pour les utilisateurs des ordinateurs de cinquieme generation. Le parallelisme inherant des programmes concus pour un tel environnement est renforce par l'utilization des graphiques, servant a specifier les programmes en PROGRAPH. Quelques exemples de programmes PROGRAPH sont presentes. L'editeur des graphiques utilise par l'usager pour developper des programmes PROGRAPH est brievement decrit.

KEYWORDS: data flow, graphics editor, functional programming language

Although the computer community witnessed in recent years important development in the field of programming languages, all the proposals ([Ada, 79], [Backus, 78], [Ashcroft and Wadge 77] etc.) share one very important feature with the earliest contributions: textual representation of programs.

Linear texts of the program are normally created by the user, therefore there is a danger that he may unconsciously narrow his ideas about problem solution and possibly even eliminate certain concepts, like parallelism, from his considerations. This apparently occurs not only when programs are created, but also when they are debugged.

The work described here addresses this problem. Roughly speaking, the user creates a graph representing the flow of data between operations of a functional language. Since a graphical representation of programs is used, rather than a textual one, the term "prograph" seems to be more appropriate than "program".

Full credit for the idea of a functional programming language with pictorial output has to go to the group at the University of Utah, which first initiated research in this area. Their language, GPL [GPL 81], in an innovative way combines functionality with graphical representation of programs. If one would attempt to compare our language, PROGRAPH, with GPL, the following features of PROGRAPH are added to GPL:

(i) provisions for a hierachy of user definitions ("subroutines");
(ii) recursion of definitions;
(iii) high level structural control operations: WHILE and IF...THEN...ELSE;
(iv) operations allowing synchronization of control flow in a coroutine-like fashion; and last, but perhaps most important
(v) apparatus for data base applications.

A preliminary implementation of PROGRAPH is under way at the time of writing.

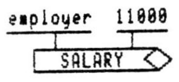
We would like to stress the fact that a prograph is by no means a flowchart. It represents flow of data, and different specific interpretations are possible, to mention pure data flow [Treleaven et al, 82b], pure demand flow [Keller, 80], and a whole spectrum of their mixtures

[Pietrzykowski et al, 82], [Treleaven et al 82a]. Any of those approaches attempts to benefit from possible parallelism.

What else is there in PROGRAPH? Brevity of this text allows us to highlight only several important concepts, without going into details. PROGRAPH draws upon both applicative (functional) languages and traditional algorithmic languages, based on van Neumann paradigm. The former contribute with the overall perspective of computational process, following "combining forms" ([Backus 78]) philosophy. The latter give PROGRAPH powerful control structures and flexibility of operational control. Moreover, PROGRAPH provides the user with a new concept of "applicative updates" which to our knowledge is a novel feature. Applicative languages leave the state of computation (meant as a memory snapshot) unchanged except for the function result, which is clearly a severe limitation in a number of applications.

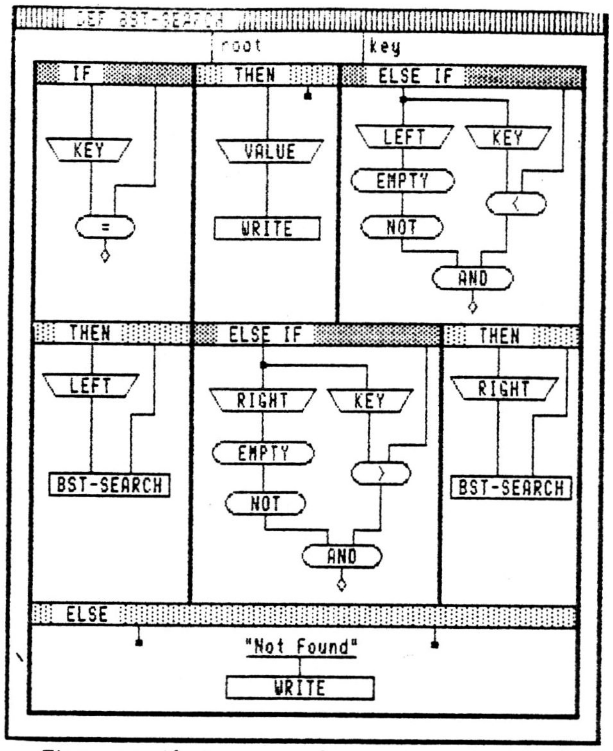
PROGRAPH approaches this issue in an entirely different way. It introduces a concept of the attribute which is partial multivalued functions defined on a set of abstract entitities (which correspond to the memory locations).

PROGRAPH provides a mechanism to find values of attributes as well as to define and modify them. The latter feature, referred to as update, is particularly important. The following simple example will illustrate how PROGRAPH handles it. Let us assume that we would like to change the existing value of the attribute SALARY on an argument (corresponding to a particular employee):



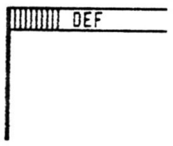
The box, surrounding SALARY, is a symbol of one of the applicative updates, which can be viewed as a combination of deletion of the attribute's value followed by the insertion of a new value. While designing PROGRAPH care has been taken to define the appropriate shapes of operation symbols, so that the user could easily associate symbol shapes with symbol meaning.

The following picture represents a prograph for binary search tree lookup.



It seems that prographs represent in a concise way a good deal of information about the structure, modularization and flow of data in the algorithm. However, the reader may wonder about the effort needed to create a picture of that kind. Therefore, an important feature of our system is the graphics editor to create graphical representation of prographs.

Suppose that the user is about to start drawing the prograph of BST-SEARCH shown earlier. He would like to begin with the outermost box. Does he have to draw all the sides of this box? Clearly, it would be too tedious a job; in our system it is enough to enter a two-letter command which will then draw the upper left corner:



Moreover, the user will be prompted to enter the name of the box. This name will automatically fill the heading of the box. Once the internal structure of the box is completed, the user may enter another command which "closes" the box, i. e., extends (or contracts) the existing corner and draws the remaining sides.

In order to create an IF...THEN box the user enters a dedicated command and types IF as a box name. The corner (with IF and THEN in its heading) is then created and the user specifies contents of both IF and THEN parts using the appropriate editor commands. Afterwards, several options are open: the box may be closed (as above), or ELSE/ELSE-IF part may follow, either to the right of the THEN part, or below it.

IF...THEN is an example of a box with a standard rectangular shape; other standard shapes are generated by dedicated commands.

To draw the wire which connect boxes, our system takes advantage of the PERQ's mouse. Using the mouse the user specifies the start and end points of the wire, and the wire itself is drawn automatically. If the wire is "bent" (at a right angle), the user has to indicate only every other bending point.

One of the design objectives of the editor was to minimize the necessary amount of typing. Command menus can be displayed on the screen and the choice of commands can be done with the mouse.

Yet another advantage of graphical representation of prographs is a new and attractive approach to debugging. The idea is to trace data flow through a selected data path by means of a token, moving along wires. Once a source of error has been discovered, a backtracking trace can be initiated. Let us also notice that such a "motion picture" of prograph execution may be used as a teaching aid, helping students to understand data/demand flow rules. Furthermore, a specific token representation can be used to indicate the type of data moving along the wire.

The work described here opens a number of interesting research problems. To highlight a few of them:

(i) optimization of execution by determining a right mix of data-flow and demand-flow policy for concurrency;
(ii) selective access of parts of a prograph from another prograph (security);
(iii) dynamic modification of prographs;
(iv) efficient run-time data structure for the interpreter and memory management.

BIBLIOGRAPHY

[Ashcroft and Wadge 77] - E. A. Ashcroft and W. W. Wadge, LUCID: A non-procedural Language with Iteration. Comm ACM 20, 7, July 1977, pp. 519-526.

[Backus, 78] - J. Backus, Can Programming be Liberated from the van Neumann Style? A Functional Style and it's Algebra of Programs. Comm ACM 21, 8, August 1978, pp. 613-641.

[Ada 79] - J. D.Ichbiah, J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Bruckner, and B. A. Wichmann. Preliminary Ada Reference Manual. SIGPLAN Notices 14, 6, Part A, June 1979.

[GPL 81] - GPL Programming Manual, Research Report, Computer Sci. Dept., University of Utah, 1981.

[Keller 80] - R. M. Keller. Semantics and Applications of Function Graphs. UUCS-80-112. October 1980. University of Utah.

[Pietrzykowski et al 82] - T. Pietrzykowski, S. Matwin, T. Müldner. PROGRAPH: A Picture Programming Language for Data Flow and Data Base Environment. Internal Report.

[Treleaven et al 82a] - P. C. Treleaven, R. P. Hopkins and P. W. Rautenbach. Combining Data Flow and Control Flow Computing. The Computer Journal, vol 25, No 2, 1982.

[Treleaven et al 82b] - P. C. Treleaven, D. R. Brownbridge, R. P. Hopkins. Data-Driven and Demand-Driven Computer Architecture. ACM Computing Surveys, March 1982, vol 15, pp. 93-143.