A Concept for Functional Programming and Distributed Data Processing in a Local Area Network and Its Implementation

Tsvetan T. Drashansky
Faculty of Mathematics and Computer Science
University of Sofia
"Anton Ivanov" Str., 5
1126 Sofia, Bulgaria

Abstract: *In this paper we describe a new approach to the pure functional programming which is based on the use of a local area network and external files. This approach essentially increases the efficiency and the field of the possible application of the functional programs. Our concepts are implemented in the framework of a FP-like functional language named FP* through the system FP*/88N. Here we state the basic mechanisms of the different system modules which demonstrate the new ideas. The advantages of the new approach are explained and compared to those known by now. The experimental applications of FP*/88N prove the existance of opportunities to create usable applied programs in pure functional style.*

1. Introduction.

Recently, plenty of scientists point out the advantages of Backus' FP systems. Backus' type functional languages posses the positive features of any pure functional language. Besides that they have some additional properties ([Bac 78], [Bac 81], [Veg 84]) : the lack of an argument in the programs permits creation of an algebra of programs as well as setting up the way of thinking of the programmers on a new higher level - relations between functions and not between objects. An important peculiarity is their relative simplicity, which makes them easy to learn and use. Some of the possible interesting applications of a system based on such a language are database systems with an amorphous structure, word processing systems, information systems, etc.

The original version of the FP-like languages [Bac 78] has some significant shortcomings - inconvenient syntax for the creation of large programs, the ability to work with external storage is not available, there are no opportunities for writing interactive programs. In [Rad 87] A. Radensky offers a programming language FP* which supports nondeterministic programming, lazy evaluation, processing of infinite data objects. FP* is

an extention of FP systems.

In this paper we suggest a concept for functional programming with distributed data processing in a local area network and relatively free use of all external devices in terms of the language FP*. The implementation of this concept in programming system FP*/88N permits the creation of a large number of programs in an almost pure functional style. The system gives means for an entirely free operation with data on files and other computers. These means are very easy to be used in programming and at the same time they have better opportunities in using external devices than the known to us systems for pure functional programming. For example, there are no functions for input and output (with side effects) in FP*/88N.

We developed experimental programs on the basis of the system FP*/88N. The most interesting of them are the distributed filer and the information system FP_INFO. We would like to stress on the fact that the changes which were necessary for the implementation of the concept do not affect the syntax of the language FP* and thus its good properties as a pure functional language. In this way the distributed filer and the system FP_INFO are relatively large programs of such class (written in a pure functional language) which can be applied in reality.

We consider also as an advantage the following feature of FP*/88N. In some similar systems the programmer has to pass commands to the system, while his program is executed (or enter such commands in the program) to manage the usage of files and external devices. There is no need of commands for that purpose in our system. The programs are sufficient for full control of data streams and external devices.

In general, our concept gives to that pure functional language means for using most of the resources of modern hardware and thus reveals the expressive power of pure functionalism on von Neumann type computers.

The programming system FP*/88N, the distributed filer and the information system FP_INFO permit to carry out experiments on real nondeterminism and parallel execution of pure functional programs.

2.Some information about the language, the system and the environment.

The language FP* is a Backus' type functional language. It is discribed in detail in [Rad 87]. FP* is designed for processing lists.

Lists are sequences of elements. Every element may be an atom or another list. The numbers, the characters and the identifiers are atoms. A special object is the so-called 'undefined object' which is denoted by '?'.

It represents, in general, the result of an unsuccesfull computation. Examples for lists in terms of FP* are:

<a b c> <<34 aaa sss <'a' 'b' 'c'>>> <>

Programs in FP* consist of function definitions. Functions always have one argument. (This is not a restriction, because we can form a list consisting of all **n** arguments of an **n**-argument function and this list will be the single argument of this function in FP*.) The argument is not written in the programs. So the function definitions include names of primitive or defined by the user functions and functional forms. The most of Backus' denotations ([Bac 78], [Bac 81]) remain valid in FP*. But here a function **f** applied to an object **x** is denoted by **x : f**. So the composition $f_1 : f_2$ means 'f_2 is applied after f_1' [Rad 87]. These denotations are more convenient for program building than the original ones.

The programming system FP*/88N includes a compiler which takes the source programs and produces a code in abstract machine instructions, and an interpreter which executes that code. (Their old versions are described in [Rad 87] but the opportunities for the usage of files are not functional and not powerful; there are no means for distributed data processing.) The compiler and the interpreter are written in TURBO Pascal. The compiler works on a IBM PC/XT/AT computer and compatibles. The computer system in which the interpreter works, represents a local area network consisting of IBM PC/XT/AT computers connected with the adapters of a MULTILINK LAN. The topology of this LAN is a ring. A copy of the interpreter works on every computer in the network which executes a specified function (on different computers they may be different). The execution represents an application of a compiled function **f** to a given object **x**. The user specifies the name of the function and then inputs its argument. Parts of **x** or the entire one can be received from the other computers in the network or from files in the external storage. The result from the application may be sent to the screen, to a file or to other computer(s).

3. The concept and its implementation.

The basic ideas of the concept are concentrated in the work of the interpreter which we shall consider now.

Suppose that the user has specified for execution (application) the function **f**. **f** has to take its argument - **x**. We shall not consider the cases in which **x** is an atom because they are of no interest. Let **x** is a list and it appears as

$$x = \langle x_1\ x_2\ \ldots\ x_n\ \ldots \rangle$$

The objects $x_1,\ldots,x_n,\ldots$ (the elements of **x**) we shall call *objects on first level of inclusion in* **x**.

In order to begin the application **x** has to be presented in an appropriate way in RAM. We can imagine that **x** is presented as a sequence of its elements $x_1,\ldots,x_n,\ldots$. In order to start the application, however, it is not necessary all the elements of **x** to be presented in RAM (in FP* they can be infinitely many). Even x_1 may be enough. This is a side of the principle of lazy evaluation accepted here. According to that principle every action in the system is started when its result is of immediate necessity (i.e. the application of the function can't continue without this result). Every action is interrupted when its result is not necessary at that moment. Here 'action' means not only evaluation but also a data input. Hence, every x_i is placed at the disposal of the function **f** when it is needed.

Let us consider where the current x_i is taken from (fig. 1). Data enters the computer through 3 channels in the described environment:

- from the keyboard;
- from other computers, i.e. from the computer serial port RS 232 connected to the network adapter of that computer
- from files in the external storage

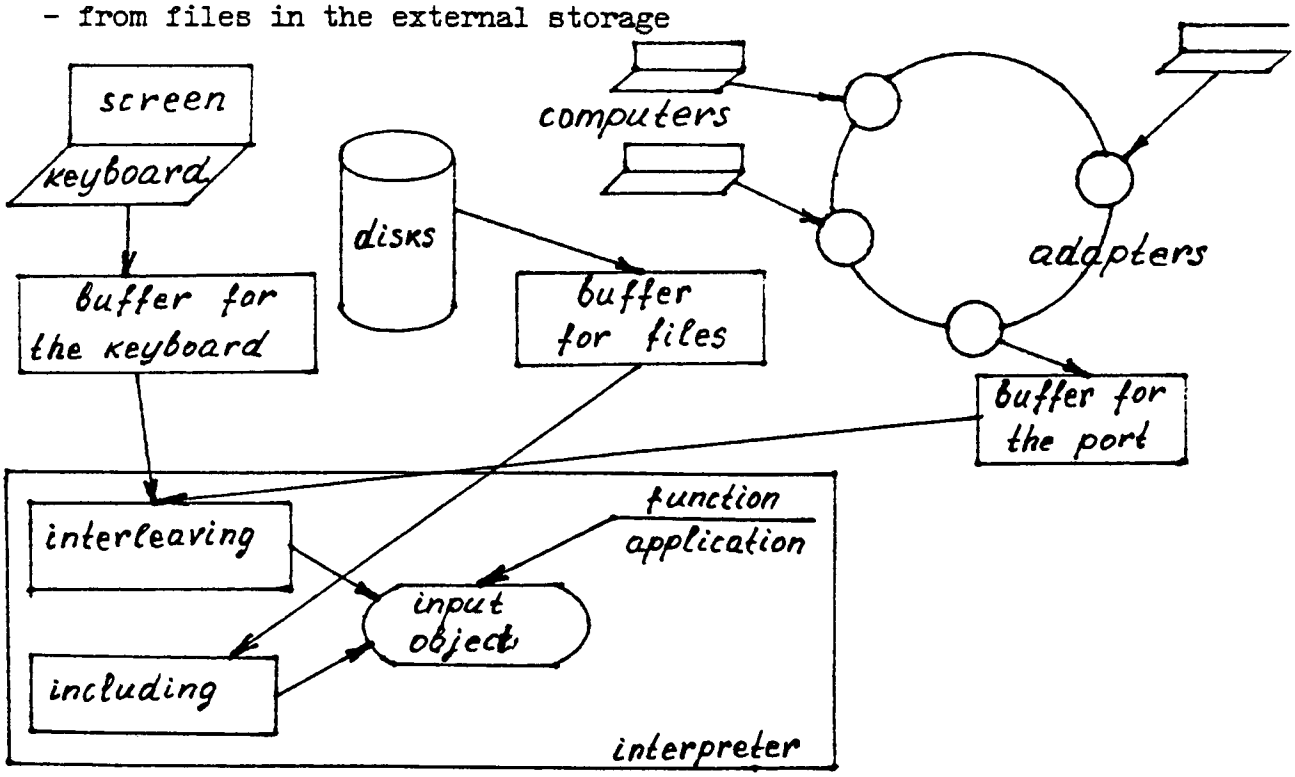

fig. 1

We shall concentrate ourselves on the first two channels. An information is received from them unpredictably in time. The interpreter consists of a few programs working pseudoparallelly (in a time-sharing manner). Two of them take the data comming from the channels and store it in two cyclic buffers in RAM. A system of pointers and control characters structures the information in these buffers in complete objects (in the sense of FP*). When the interpreter has to input the current $\mathbf{x}_i$ it takes a complete object from where there is such one. For example, if the buffer for the keyboard contains

<<2 3> a b> 145 fff....

and the buffer for the port contains

< <45 67 a> b c ...

then next $\mathbf{x}_i$ will be <<2 3> a b>, because the list in the port buffer is not complete yet. If $\mathbf{x}_i$ is taken from the port buffer it appears as

$\mathbf{x}_i$ = <**from_comp** *name_of_sender data*>

The lack of the predefined identifier **from_comp** as a first element of $\mathbf{x}_i$ means that $\mathbf{x}_i$ is taken from the keyboard buffer. The function **f** can choose the necessary information (through the means of a filter written in FP*) and form its argument (if, for example, **f** needs data comming only from one place).

Thus, a nondeterministic interleaving of two data streams is obtained through the described method. This interleaving is performed always and independently of the contents of the function **f**, but the user can write some elementary functions in FP* which filtrate the interleaved data stream.

When $\mathbf{x}_i$ is placed at the disposal of the function **f**, the input of $\mathbf{x}_{i+1}$ is delayed until it is needed. (Obviously, both buffers - for the keyboard and for the port are filled with data continuously.)

In general, if we don't take into account the third channel, **x** consists of elements on first level of inclusion, which are input from the keyboard or from the port according to their chronological arrival.

From the third channel the data is included in the input stream by request of the user. Every file in FP* consists of only one list, that is, its format is amorphous, completely corresponding to the data format used in FP* programs. If the programmer includes in his program the construction

<'**from_file** *filename*>

then he may think that instead of that expression in the processed object stands the list from the file named *filename*.

Such a construct may be included in the program (and thus in the input stream) on an arbitrary level of inclusion. Let us note that the remaining part of the current input object (on first level) is input from the initial channel. The input of the objects on the first level of inclusion of the file is also delayed, so the file contents is read only and until it is necessary. As we see, data stored in files are included in the input stream easily, with elementary programming means. This method in combination with the list processing instruments in FP*, however, offers to the programmers almost unrestricted opportunities for using information from files.

Let us consider now the output data stream of the interpreter. In fact, the output data stream is the result **y** of application of the function **f** to the object **x**. Let **y** be a list. **y** is constructed by the function, i.e., by the programmer. **y** can consist of lists on an arbitrary level of inclusion. Every list can be sent through one of the following channels:

- the screen;
- the port;
- to an external file.

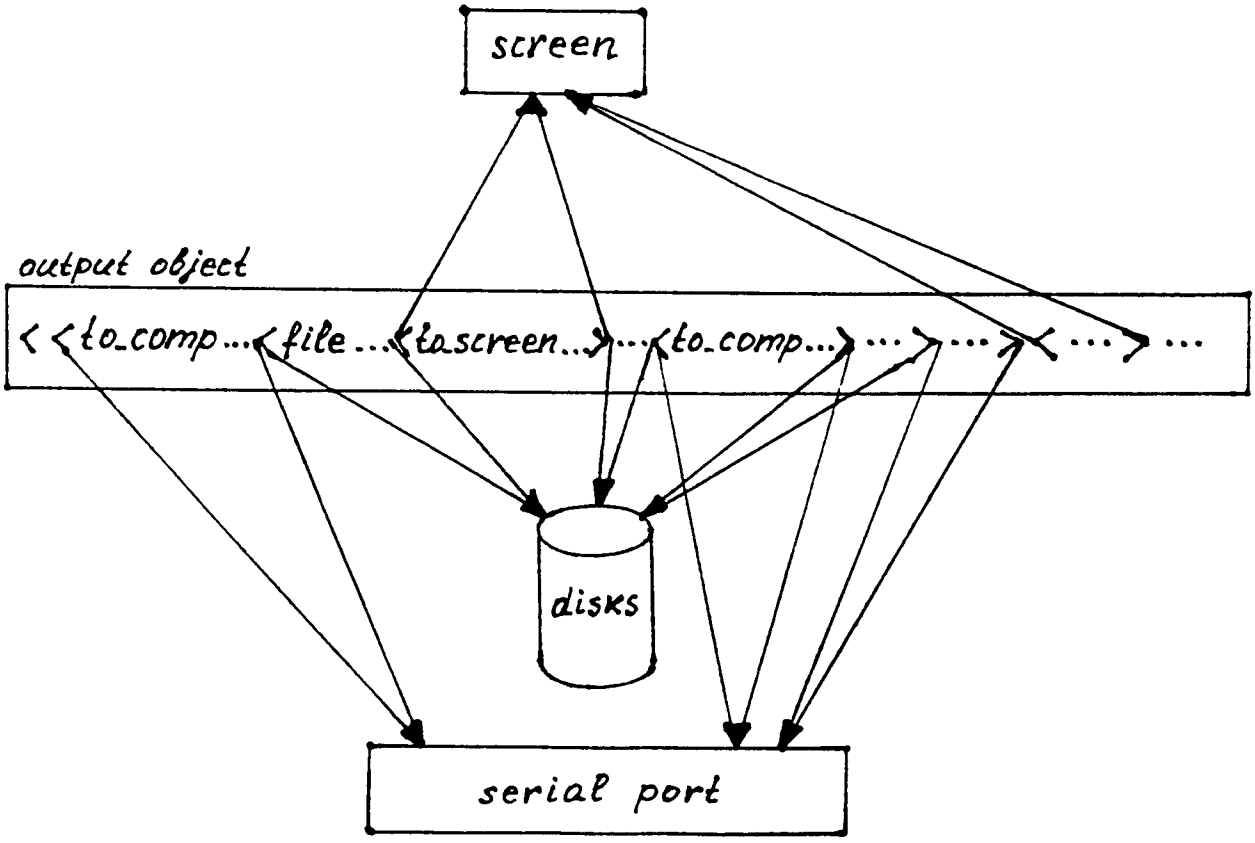

fig. 2

Data can be distributed to the places that are addressed (fig 2). For example, through the third channel data may be stored in different files on different devices; through the port information may be addressed to different computers and so on. Besides, if a given list is addressed to a certain destination, every included in it list can be sent to a different place. The distribution is made in the following way:

The identifiers **to_screen, to_comp, to_file** and **file** are predefined in FP*/88N. If the list z is a part of the input object and its first element is one of those identifiers, then z will be sent correspondingly to the screen, to a specified (as a second element of z) computer, will be added to a specified (as a second element of z) file or the contents of z will be stored as a new file. After z is sent, the output continues as was specified before. If the first element of z is not a predefined identifier, then z is sent to the same place as the list in which z is included. The data portions intended for different destinations can be included into oneanother in arbitrary manner.

The fourth possible output channel - the printer, is used to duplicate the information, destinated to the screen (by the user's request).

This functional method for a program results distribution gives the programmers quite powerful means for the structuring of processed data. But it leads to some conflicts. We would like to mention some of them. Splitting of data portions, on one side, and the peculiarities of operations with the output channels, on the other side, impose the joining of the portions into different buffers and their sending in an appropriate way. In fact, the need of output of information triggers every function application (according to the principle of lazy evaluations). Many conflicts here are related to the case when a complete part of the output is available but the input from the destination of this output is not finished. All those conflicts are successfully solved by the system.

4. Summary.

Our concept can be applied to any similar to FP* language or to FP* in arbitrary environment but we explain it here on the basis of FP* and in the environment described above.

The described approach permits the creation of entirely usable almost pure functional programs. These programs are very short - for example, the distributed file system occupies 3 pages of source. Because of the file structure in FP*, one can build database or information systems in which

data is stored and processed independently of its structure and format and, hence, any new information can be easy received in these systems.

When we use distributed data processing in a local area network we can easily (with simple programs) exchange messages among the users, use/store information from/in external storage of other computers, protect any data in our own files, etc. The major advantage of our system is that it permits an easy creation of working and applicable distributed functional programs from such a wide field.

5. References.

[Bac 78] Backus,J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Comm. ACM*, Vol. 21, Aug. 1978

[Bac 81] Backus,J. "The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions." *Proc. Int. Colloq. on Formalisation of Programming concepts, Peniscola, Spain*, LNCS, Vol. 107, Heidelberg, Springer Verlag, 1981.

[Veg 84] Vegdahl,S.R. "A Survey of Proposed Architectures for the Execution of Functional Languages", *IIIE Trans. on Computers*, Vol. 12, Dec. 1984.

[Rad 87] Radensky,A. "Lazy Evaluation and Nondeterminism Make Backus' FP-systems More Practical", *SIGPLAN Notices*, Vol.22, N.4, Apr. 1987.