LAZY EVALUATION AND NONDETERMINISM MAKE BACKUS' FP-SYSTEMS MORE PRACTICAL

Atanas Radensky
Center for Mathematics and Mechanics
1090 Sofia, P.O.Box 373, Bulgaria

SUMMARY. Backus' FP-systems are made more practical by introducing into them lazy evaluation and nondeterminism. This is done in the framework of a concrete programming language called FP*. From the one hand, this language is almost as mathematical as FP-systems are. From the other hand, it gives the possibility to manage secondary memory and to develop such applications as, for instance, interactive and distributed file systems. Experimental versions of a compiler and an interpreter for the FP* language are implemented.

1. FP-SYSTEMS - SOME SHORTCOMINGS

J. Backus, in his Turing award lecture in 1977, introduced a class of pure functional programming languages, referred to as FP-systems [Bac, 78]. In FP-systems, only functions of one argument are used. (Since the argument is just one, there is no need to explicitely denote it, and this is the reason FP-programs do not contain any variables.) This results in a very simple and elegant mathematical semantic, which encourages the development of a peculiar algebra of programs [Bac,78]. Some general theorems which characterize large classes of programs are proved [Wil,80], [Bac,81], [Wil,82]. For instance, some theorems concerning the representability of FP-functions in the form of "infinite conditions" are proved: the recursion and iteration theorems [Bac,78], the linear expansion theorem [Bac,81], the theorem of representation of some non-linear forms [Wil,80], the theorem which establishes the existence of representation of any function in the form of infinite condition [Rad,82].

The possibility to develop relatively simple and powerful mathematical methods for formal study and analysis of programs is one major advantage of FP-systems. However, there exist some hard shortcomings of FP-systems which are obstacles to their wider spreading and use as a real programming tool.

First of all, FP-systems do not support the development of interactive programs and the management of secondary storage. Thus, it is impossible in FP-systems to implement text editors, file systems and database systems, and many other similar programs.

Another group of problems with FP-systems concerns the efficiency of their implementation. It is still unclear how much combinators or multiprocessor systems can help in this respect. Much experimentation ought to be done, in order to develop specific compiler techniques for efficient code generation.

The above mentioned and some other shortcomings of FP-systems can be removed (at least partially) by means of specifically adopted variants of some well known methods: lazy evaluation, nondeterminism, local definitions, tabulation. Such an approach to improving applicability and efficiency of FP-systems is materialized in a concrete language called FP*, and a corresponding microcomputer programming system. From the one hand, FP* succeeds useful mathematical properties of FP-systems. From the other hand, FP* allows interactive programming, secondary storrage management, and it can be implemented efficiently enough in order to be considered as a

candidate for real programming.

2. BASIC CUNSTRUCTS IN THE FP* LANGUAGE

Williams [Wil,82] presents a very nice synopsis of FP-systems, and here we use its scheme to present the FP* language, the more so as many constructs in FP* are the same as in FP-systems.

A program in FP* is a set of definitions of functions that map objects to objects. The set Ob of objects is defined in the following way:

1) ? is in Ob (? denotes the undefined object);

2) a set of atoms (integers, identifiers, and characters) is in Ob;

3) any finite or infinite sequence <x1 x2 ... xn ...> of objects is in Ob, including the empty sequence <> (a precize definition of infinite objects can be given in terms of infinite trees).

Backus denotes the undefined object by $\perp$ and accepts that $\langle x1,x2,\ldots,xn\rangle = \perp$, if for some i, $xi = \perp$. Such strictness of the list constructor in FP-systems implies that "...the various computation rules, outermost, leftmost-outermost, innermost, and leftmost-innermost, are all safe rules for computing the least fixed point" [Wil,82]. This is really very nice from a theoretical point of view but is not so important from a practical one. The price which is paid for the strictness of the list constructor in FP-systems is the absence of lazy evaluation, and this price is too high.

A nonstrict list constructor is used in FP*. In particular, this means that the object <...?...> is not equivalent to the undefined object ?. This assumption allows lazy evaluation in FP*. Note that with nonstrict constructor it is still possible to prove laws in the algebra of programs, use structural induction and fixpoint induction to deduce properties of programs, and in general - reason about programs by means of precize mathematical methods.

A set of primitive functions and a set of functionals (referred to as "functional forms") are used. Given a functional program P, the notion of a function (programmable in P) is defined as follows:

- Any primitive function is a function (programmable in P).
- If f1, f2,..., fn (n >= 0) are functions (programmable in P), and if C is a functional form that take n arguments, then C(f1,f2,...,fn) is a function (programmable in P).
- If the program P contains a definition

lazy g = G(g,...),

then g is a "lazy" function (programmable in P). If P contains a definition

def f = F(f,...),

then f is a "normal" function (programmable in P). The right sides G(g,...) and F(f,...) are functions (programmable in P).

Further on, the expression "programmable in P" is omitted. By x:f we denote the result of the application of f to x.

When a lazy function is to be applied to an object x, a pseudo-object which replaces the result x:f is created. In that, the corresponding computational process is delayed, not initiated. When the result x:g of the application of g to x is to be output or used by a primitive function, then the suppressed computational process is automatically forced.

The set of primritive functions contains selector functions (denoted by 1, 2, 3,...), the tail function (tl), functions-predicates (=, is_empty, is_atom, eq0, etc.), arithmetic functions (+, -, *, /, mod, plus1, minus1), the append functions (apndl, apndr), and many others; some of them are considered later.

The following examples highlight the semantics of primitive functions used further in this paper:

Selectors: <a b c>:1 = a; <a b c>:3 = c;

Tail: <a b c>:tl = <b c>; <a <b c>>:tl = <<b c>>;

Identity: <a b c>:id = <a b c>;

Is_empty: <a b c>:is_empty = F; <>:is_empty = T; ?:is_empty =?;

Multiplication: <100 2>:* = 200; <100 ?>:* = ?.

The append to the left function (it differs essentially from the corresponding FP-function) is applied in the following way:

x:apndl =
<y>, if x=<y <>...>;
<y z1 z2...zn>, if x=<y <z1...zn...>...>;
<y>, if x=<y z...> and z is not a list;
? otherwise.

The set of functional forms includes composition (denoted by ":", not by "o" as in FP-systems), construction ([f1 f2...fn]), condititon (if p then f else g), constant (denoted by 'x, not by $\overline{x}$), iteration. A "case" functional form is introduced in FP* in order to improve both program readability and efficiency. This form will be included in some further examples.

According to Backus [Bac,78], functions in an FP-composition (f1 o f2 o ... o fn) are applied from right to left, i.e.:

(f1 o f2 o ... o fn):x = f1:(f2:...(fn:x)...).

This is rather inconvenient, since in real programs n can be fairly large. This definition of composition forces programmers (which usually write from left to right) to write firstly the function which is to be applied the last. (Function composition is used like statement composition in procedural languages, and we can imagine a Fortran programmer which is persuaded to write his program starting with the last statement and finishing with the first one.) In FP*, functions in a composition (f1:f2:...:fn) are applied in the way are writhen, i.e. from left to right:

x:(f1:f2:...:fn) = (...((x:f1):f2)...:fn).

Some examples concerning functional forms follow.

Composition: <a b c>:(tl:1) = b ;

Construction: <5 2>:[+ - * /] = <7 3 10 2>;

Constant: 5:([id '2]:*) = <5 2>:* = 10;

Condition: <5 6>:(if empty then id else [1 '2]:*) = <5 2>:* = 10.

3. INPUT - OUTPUT IN FP*

Suppose, a function f is applied to an object x. The application of f starts (if possible) even before any part of x is input. The application of f is suspended in a case some nessessary parts of x are not input yet. It is resumed immediately after one or more lines containing the nessessary data are input. In that, any component of the result x:f is output immediately after it has been computed.

Similarly to [Hen,82], a function which doubles an infinite list of numbers can be introduced:

lazy double = [[1 '2]:* tl:double]:apndl.

During its application, the following dialogue can be seen on the screen:

```
<1 3 2          --input
<2 6 4          --output
 5 7 10         --input
 10 14 20       --output
 6              --input
 12             --output
```

...

A primitive function "write" is introduced in FP*. It extends the primary memory onto the secondary one.

We present here a simplified variant of this function. By means of it, an object can be output onto a disk. Formally, the function is applied according to the following rule:

x:write = y, if x=<y n ...>, else ?

If n is a number of a disk device, then the object y is output onto the corresponding disk. In that, the piece of primary storage occupied by y is released. Instead, a pseudo-object which contains some nessessary disk addresses is created. In this manner, the object y remains accessible for further use, and when nessessary, is automatically loaded by the system into the primary storage.

Henderson [Hen,82] described a simple filer in a Lisp-like notation. We did the same in the FP* language using the primitive function write. A function "filer" is described in the FP* language. By means of this function, the user can create, update, and interrogate a database, which is actually a list of files. A file is an object in the form <file_name file_value>. A file_name is an identifier, and a file_value is an object.

The filer accepts some commands from the keyboard, executes them and returns corresponding answers on the screen. We consider here only two commands: put and get. The put command has the form:

<put file_name file_value>.

The result of the execution of this command is that the file is stored in the database, and answer "ready" is displayed on the screen.

The get command has the form:

<get file_name>.

As a result, the corresponding file_value from the database is displayed on the screen. Alternatively, the answer "not_found" is displayed, if a file with the specified name is missing from the database.

The filer is a function which is applied to an infinite list of commands, executes them, and returns corresponding answers. As an effect of lazy evaluation, the filer executes and answers every command immediately after it has been input. Thus, there is a dialogue between the user and the filer.

Example. A typical dialogue looks like this:

```
<<put file1 <A B C>>
<ready
 <get file1>
 <A B C>
 <put file1 <x y>>
 ready
 <get file1>
 not_found
 <get file1>
 <x y>
 ...
```

Note that the FP* programming system can save the state of the executed functional program before any switching off the computer; it can recover it immediately after the computer is switched on again. In that, the database is saved on the disk, and is further used and delopped after the switching on.

There is no place enough to present the definition of the filer here, but this function will be used in further examples.

4. NONDETERMINISM IN FP*

A primitive function-predicate named "is_ready" can be used in order to test whether its argument is a pseudo-object, or it is an actual (i.e., already computed) one. The value of the function can be T or F, and it depends on the status of the argument in the moment of application. If the argument is a pseudo-object, then the function is_ready returns F, else it returns T.

This function is nondeterministic since it can return different results when applied twice to the same argument. For instance, the result of application may depend on the speed with which the argument is input.

Formally, it can be accepted that the value of the function is_ready is produced in an absolutely random manner:

x:is_ready = T or F.

One more primitive function named "await" is introduced. Its application is tightly connected to the use of the function is_ready. Formally, the function await gives just the same result as the identity function:

x:await = x.

In practice, the implementation of await is different from the implementation of the identity, since the former function forces its argument. Thus, the application of await to a pseudo-object resumes some delayed computational process.

Further, some definitions of lazy functions are considered. They illustrate the use of the functions is_ready and await.

A NONDETERMINISTIC MERGE OF TWO LISTS

A nondeterministic function "merge" is defined, which when applied to an object in the form

<<x1 x2 x3 ...> <y1 y2 y3 ...> ...>,

merges <x1 x2 x3 ...> and <y1 y2 y3 ...> in the same list:

```
lazy merge =
  if 1:is_ready then
    [1:1 [1:tl 2]:merge]:apndl
  else if 2:is_ready then
    [2:1 [1 2:tl]:merge]:apndl
  else [1:await 2:await]:merge;
```

Henderson [Hen,82] proposes as a primitive function in a Lisp-like language a function called "interleave" which is applied analogously (but not equivalently) to the function merge. The use of the primitive functions is_ready and await instead of interleave however gives the possibility to describe various useful variants of interleaving functions which fit better to different particular tasks.

Further, lists in the form:

x = <<i1 x1> <i2 x2> <i3 x3> ...>

are considered, where i1, i2, i3, ... are integers and x1, x2, x3, ... are arbitrary objects. The integers i1, i2, i3, ... are referred to as labels, and x is referred to as labeled list.

Some functions which filter labeled lists can be defined. These functions filter a list, letting pass only elements with particular labels. In that, filters remove labels from all screened elements.

In further examples,only two filters named filter01 and filter02 are used. Filter01 (respectively filter02) lets pass only elements labeled by 0 or 1 (respectively by 0 or 2). The filter functions are lazy, and are defined by means of the "case" functional form:

```
lazy filter01 =
  case 1:1 is
```

```
        'O, '1  --> [1:2 tl:filterO1]:apndl;
        else tl:filterO1;
      end;
```

FilterO2 is defined analogously.

Example. Denote by W the following list:

<<0 100> <0 200> <2 300> <1 400> <0 500> <1 600> <2 700> <1 800> ...>.

Then:

W:filterO1 = <100 200 400 500 600 800...>,

W:filterO2 = <100 200 300 500 700...>.

SPLITTING, PROCESSING, AND MERGING

Suppose, the following process (see fig.1) is to be programmed:

1) Elements of a labeled list are input one by one from the keyboard;

2) The input data are filtered and split up on two separate lists;

3) Every element of the first (respectively - the second) list is processed by means of a function g1 (respectively - g2);

4) The both processed lists are merged and the resulting list is output on the screen.

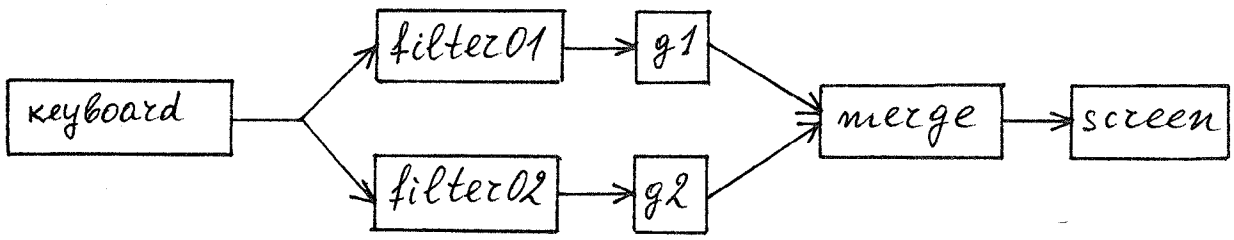

Fig. 1.

This process is implemented by means of a function defined as follows:

```
lazy p = [filterO1:g1 filterO2:g2):merge.
```

Clearly, various processing functions g1 and g2 can be used. For instance, g1 and g2 can be defined in the following way:

```
lazy g1 =
  await:
  (if is_ready then [1:plus1 tl:g1]:apndl
  else g1);
lazy g2 =
  await:
  (if is_ready then [1:minus1 tl:g2]:apndl
  else g2).
```

Thus, g1 adds 1 to (and g2 subtracts 1 from) every element of an infinite list of integers.

Note that the function merge demands (by means of the primitive function await) the result of the application of g1 and g2 (see fig.1). Actually, the function merge forces g1 and g2. From its part, g1 (respectively g2) demands the result of application of filterO1 (respectively filterO2). Thus, every function which is on the path from the keyboard to the screen demands the result of its predecessor.

A DISTRIBUTED FILER

We are going to define a distributed filer which creates and manages some databases in parallel. The distributed filer will be able to execute the following commands:

1) <i <put f x>>.

If i>0, then this command has the effect of adding the file <f x> to the i-th data base. In case of i being 0, the same is done with all data bases. If a file with name f already exists, then it is replaced by the new file.

2) <i <get f>>.

If i>0, then this command extracts from the i-th database the value of the file with name f. In case of i being 0, file values with name f are extracted simultaneously from all databases.

In fact, the above commands are labeled variants of the commands put and get, considered in section 3. Label 0 means that the command is to be executed on all databases, and label i, i>0, means that the command is to be executed on the i-th database.

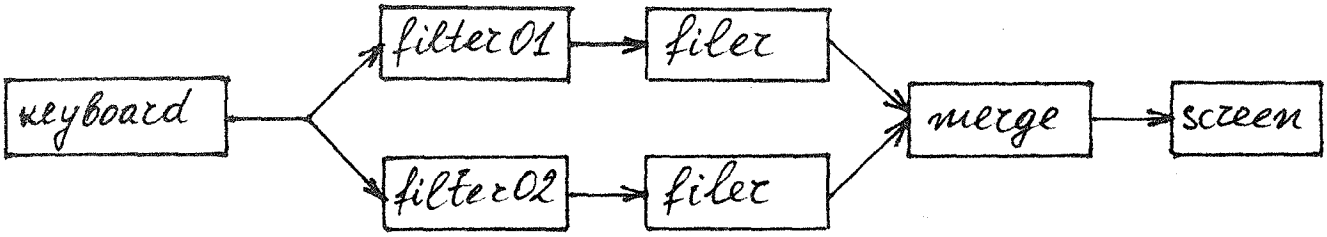

Fig. 2.

A function named distributed_filer is defined below. For simplicity, it supports only two databases (see fig.2):

```
lazy distributed_filer =
        [filter01 : filer   filter02 : filer] : merge ;
```

In the above definition, the filer that supports one database (see section 3) is used.

The disributed_filer is applied to an infinite sequence of commands, executes them, and returns an infinite sequence of answers. Due to lazy evaluation, every command is executed and answered immediately after it has been input. A typical interaction with the distributed_filer looks like that:

```
<<1 <put f 100>>
<ready
 <2 <put f 200>>
 ready
 <0 <get f>>
 100 200
 <2 <get f>>
 200
 ...
```

In the begining, a file <f 100> is stored in the first database, and a file <f 200> - in the second one. Actually, f can be considered as the same file, distributed in the both databases. By means of the command <0 <get f>>, the file f is extracted simultaneously from the both databases.

In practice, a distributed filer which supports much more than two databases can be used. It is convenient to implement such a function (and many others) by means of multiprocessor, multidrive systems, in which merging is done by hardware. In particular, it is convenient to allocate every database on a separate intelligent unit which comprises a disk drive, a processor, and some primary memory.

5. CONCLUDING REMARKS

An experimental programming system FP* is implemented on the Bulgarian microcomputer "Pravets-82". It comprises a compiler which

translates functional programs into abstract machine language programs, and an interpreter of such abstract programs. A number of functional programs are tested and executed, including all programs considered in this paper.

Let us finally note that management of text databases is considered as a major application of the FP* language. It can be efficiently supported by means of a multiprocessor multidrive computer system.

6. REFERENCES

[Bac, 78]
J.Backus, Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs. Commun. ACM, v. 21,Aug.1978, pp. 613-641.

[Bac, 81]
J.Backus, The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions. In "Proc. Int. Collog. on Formalisation of Programming Concepts", Peniscola, Spain, Lecture Notes in Computer Science, vol.107 (Springer Verlag, Heidelberg, 1981).

[Hen, 82]
P. Henderson. Purely Functional Operating Systems. In "Functional Programming and Its Applications", ed. by J. Darlington, P. Henderson, and D. Turner, Cambridge Univ. Press, 1982.

[Rad, 82]
A.Radensky, An Infinite Expansion in Iterative Combinatory Spaces and in Functional Programming Systems. Comptes rendus de l'Academie Bulgare des sciences, Tome 35, N 5, 1982, pp.569-571.

[Wil, 80]
J.Williams. On the Development of the Algebra of Functional Programs. Tech. report RJ 2983, IBM Research laboratory, San Jose, CA, 1980.

[Wil, 82]
J.Williams. Notes on the FP style of functional programming. In "Functional Programming and Its applications", ed. by J. Darlington, P. Henderson, and D. Turner, Cambridge Univ. Press, 1982.