Computers and Artificial Intelligence, Vol. 11, 1992, No. 2, 375—384

ON GENERATING FUNCTIONAL PROGRAMS FROM PROLOG SPECIFICATIONS

Vishv Mohan MALHOTRA

Division of Computer Science, Asian Institite of Technology, GPO Box 2754, Bangkok 1051, Thailand

Anurag JAIN

Citicorp Overseas Software Ltd., 133/SDF V SEEPZ, Andheri (E), Bombay 400096, India

Abstract. We identify a class of PROLOG programs that can be used as the specification of the functional programs. Using the mode information, the functions needed to construct the functional program are identified. The function definitions may then be derived from the clauses in PROLOG specification. Rules have been given for constructing a FP-like definition for the function representing the user query.
The methodology makes an extensive use of the algebra of functional programs to simplify and consolidate these representations. The methodology has been illustrated using an example.

Keywords: logic programming, functional programming, program transformation.

1 INTRODUCTION

A *logic program*, for instance a PROLOG program, states the problem succinctly and in an easy-to-understand form because it states *'what to compute'* without also stating *'how to compute'*. The absence of the control information, however, requires that the programs be executed *non-deterministically*. The non-deterministic execution of the PROLOG programs on a sequential machines is usually achieved through a search involving *backtracking*. This leads to runtime inefficiency and overheads. In general, one may identify portions in the execution trace that were discarded and did not contribute to the end results due to backtracking.

Functional programming languages such as FP [1], on the other hand, incorporate adequate control information and are amenable to efficient execution. Backus has shown that these languages have well defined *algebra* that can be used to *transform* functional programs from one form into another. The algebra can also be used to prove many

interesting properties of the programs. Indeed, the functional programming paradigm is central to even a well-structured program in an imperative language.

In this paper we exploit an algebra of a suitable extension of FP to transform a class of PROLOG programs into functional programs. This will provide us a method for constructing functional programs from *specifications* expressed in a restricted class of PROLOG. Some other works on translation of PROLOG programs into functional programs have been reported in [4, 11].

Consider, for an example, the following PROLOG program taken from [4]. Given a string s and an integer i, the program returns a prefix, p of s of length i.

$? \leftarrow \text{Front}(i, s, p)$.
$\text{Front}(i, s, p) \leftarrow \text{Append}(p, r, s),\ \text{Length}(p, i)$.
$\text{Length}([\,], 0)$.
$\text{Length}([h|t], i) \leftarrow j := i - 1,\ \text{Length}(t, j)$.
$\text{Append}([\,], r, r)$.
$\text{Append}([h|t], r, [h|u]) \leftarrow \text{Append}(t, r, u)$.

The program non-deterministically chooses the prefix, p of s and verifies that p is of length i. Thus, the execution of the above program will make several (in fact, $(i + 2)(i + 1)/2$) calls to the procedures Append and Length. We will eliminate the non-determinism by first converting the program into a functional form and then using the equalities of the program algebra to generate an efficient program for Front.

In Section 2, we characterize the PROLOG programs which may serve as the initial specification of the problem to which the methodology developed here can be applied. We also state briefly the main steps in converting a PROLOG program into a functional program. In Section 3, an extension of FP called Non-deterministic FP (NFP) is introduced and laws of its algebra are given. In Section 4, using the program for Front as an example, we describe a procedure for converting PROLOG programs into non-deterministic functional programs. In Section 5 some simple rules for consolidating multiple definitions for a function into a single definition are given. Section 6 sums up the present status of the work and points the directions of the continuing work.

2 CHARACTERIZATION OF PROLOG PROGRAMS

The relationships between the logic programs and functional programs have been studied extensively [4, 11]. It may be suggested, based on these studies, that a PROLOG program can be used as a specification for a functional program if it is *directional* and satisfies the *functionality* requirements. Both these issues have been investigated in literature. Several algorithms based on the *mode analysis* are known for determining the directionality of a PROLOG program [6, 10, 11]. The functionality problem, however, is *undecidable* [5].

A program, P, is directional if all predicates in the program can be assigned one or more modes such that every call to the predicate (procedure) satisfies one of the assigned

mode constraints. A mode on a predicate partitions the arguments of the predicate into *input* and *output arguments*. It is required that the input arguments of a predicate be *ground terms* and no variable in any of the output argument positions be *instantiated* when a call to the (predicate) procedure is made.

We associate a *relation* (in relational database [3] sense) with each predicate in the program. The *attributes* in the relation correspond with the argument positions in the predicate. Every tuple in the relation is *ground* and satisfies the predicate. Let p be a predicate and R be the corresponding relation. We say P satisfies the functionality criterion under a mode M if the input columns under M *functionally determine* the output columns in R.

As mentioned earlier the functionality criterion is undecidable — there does not exist an effective algorithm to test this property. It is, however, of interest to mention that some examples follow showing the sufficient conditions for a query (predicate with mode) to be functional in a PROLOG program:

(i) all predicates under each of the assigned modes are determinate; that is, for a given input they can satisfy at most once. Or
(ii) every predicate under the assigned modes in the program satisfies the functionality criterion.

It should be clear, however, that a query can satisfy functionality criterion without all predicates in the program being either determinate or functions. For example, Front (input, input, output) is functional, though Append (output, output, input) is not. Note that the procedure Append is called in the above mode while generating the prefix of a string.

As a result of undecidability of the functionality criterion our method is necessarily incomplete. As a consequence, it can be augmented to encompass bigger and more ambitious problems. However, we believe that even a small set of rules allow the method to develop many interesting functional programs. The main steps in transforming specifications into functional programs are:

(i) For each predicate in the PROLOG program, identify the modes in which it will be called during the execution of the program.
(ii) Assign a unique function name to each predicate-mode pair.
(iii) Using the procedure to be explained in Section 4, convert **PROLOG** clauses into non-deterministic definitions of the functions.
(iv) Remove non-determinism and consolidate the NFP definitions into a function defining the user query.

Table 1 gives the modes and the function-names assigned to the various predicate--mode pairs of interest in the execution of the Front program.

Tab. 1. Modes of various calls in the execution of the Front program

PREDICATE	MODE	FUNCTION-NAME
Front	input, input, output	Front
Append	output, output, input	Unappend
Length	input, input	Length
:=	output input	built in

3 A NON-DETERMINISTIC FUNCTIONAL PROGRAMMING LANGUAGE — NFP

NFP is a simple extension of FP [1]. We introduce the language by pointing out its main differences from FP. A NFP function may have several definitions. Sometimes, we refer to these multiple definitions of a function as *fragments* of the function definition. Each use of a function that admits *multiple definitions* is prefixed with a special operator, #. A function is said to admit multiple definitions if it is defined more than once, or, if its definition contains a function that admits multiple definitions (i.e. it has a # prefixed to it). The operator # is *non-deterministic choose operator* (oracle) and is responsible for choosing the correct fragment defining the function. For an easy identification, we shall introduce fragments using the $\approx$ symbol; a function that has only one definition will be defined using the $\equiv$ symbol. The distinction is useful when *folding* [2] function on its definition. Only the definitions introduced through the $\equiv$ symbol may be used for this purpose.

Another change in the definition of FP has been made to adapt it to the PROLOG like environment. The object F (False) also denotes the *undefined object*, $\perp$. Any function returns F when applied to an object F.

With some obvious changes the laws of FP algebra continue to apply to NFP, too. In particular, any law that has one occurrence of a function on one side of the equality and more than one occurrence on the other side is not valid if the function admits multiple definitions. The difficulty arises because each of these occurrences can be replaced by a fragment of the function independent of one another. We restrict the use of the algebraic laws only to the cases satisfying the following condition: *there should not be more than one occurrence of a function that admits multiple definition on either side of the equality.* If f admits multiple definitions, the following law is not valid: $[1,2] \circ \#f = [1 \circ \#f, 2 \circ \#f]$. However, $1 \circ [1,2] \circ \#f = 1 \circ \#f$ is valid.

We list below some of the equalities of NFP. This list, in fact, suffices for all our examples in this paper. In the following list of the equalities, n and s denote selector functions, and u denotes a function that does not admit multiple definitions. Other symbols represent functions other than $\bar{F}$.

T1	$s \circ [1, 2, \ldots, n, f]$	$= \bar{F}$	if $s > n + 1$
T2		$= f$	if $s = n + 1$
T3		$= \bar{T} \circ f \rightarrow s; \bar{F}$	Otherwise

T4 $f \circ \bar{F} = \bar{F}$
T5 $\bar{F} \circ f = \bar{F}$
T6 $\bar{x} \circ f = \bar{T} \circ f \rightarrow \bar{x}; \bar{F}$
T7 $\bar{T} \circ f = T$ if f is total and f returns objects other than F for all inputs other than F
T8 $u \circ (c \rightarrow g, h) = c \rightarrow u \circ g; u \circ h$
T9 $(c \rightarrow g; h) \circ u = c \circ u \rightarrow g \circ u; h \circ u$
T10 $[f1, \ldots, fn] \circ u = [f1 \circ u, \ldots, fn \circ u]$
T11 $[f \circ 1, g \circ 2] \circ [h, t] = [f \circ h, g \circ t]$
T12 $\wedge \circ [f, g] = f \rightarrow g; \bar{F}$
T13 $\bar{T} \rightarrow f; g = f$
T14 $\bar{F} \rightarrow f; g = g$
T15 $[\ldots, \bar{F}, \ldots] = \bar{F}$
T16 $(c \rightarrow \mathrm{id}; \bar{F}) \circ [f, g] = [f, (c \rightarrow 2; \bar{F}) \circ [f, g]]$
T17 $= [(c \rightarrow 1; \bar{F}) \circ [f, g], g]$
T18 $= (c \circ [f, g]) \rightarrow [f, g]; \bar{F}$

The function apply defines an interpreter for NFP.

```
function apply (f: function, x: object)      :
  begin
    case f of
      f is a primitive function               : return f(x);
      f is a defined function and does
        not admit multiple definitions        : begin
                                                  g := definition (f);
                                                return apply (g, x)
                                                end;
      f has the form # h                      : begin
                                                  g := choose_def (h);
                                                  return apply (g, x)
                                                end;
      f has the form g ∘ h                    : return apply (g,
                                                        apply (h, x));
      f has the form [f1, f2, ..., fn]        : begin
                                                  Ans := true;
                                                  for i := 1 to n do
                                                  begin
                                                    xi := apply (fi, x);
                                                    if xi = F
                                                      then Ans := false
                                                  end;
                                                  if Ans
                                                    then return
                                                        ⟨x1, x2, ..., xn⟩
```

```
                                    else return F
                                end;
    f has the form (c → g; h)   : if apply (c, x) ≠ F
                                    then return apply (g, x)
                                    else return apply (h, x)

    end /* Case statement */
end.
```

4 GENERATING NFP FUNCTION DEFINITIONS

Let M be a mode for a call to predicate P. Let $P \rightarrow P1, P2, \ldots, Pn$ be a clause. Let $M1, M2, \ldots, Mn$ be the modes of the calls to predicates $P1, P2, \ldots, Pn$ respectively in the clause. Let $\Psi, \Psi1, \Psi2, \ldots, \Psi_n$ be the function names assigned to the predicate-mode pairs $\langle P, M\rangle, \langle P1, M1\rangle, \langle P2, M2\rangle, \ldots, \langle Pn, Mn\rangle$ respectively. The clause $P \rightarrow P1, P2, \ldots, Pn$ defines the function Ψ as follows:

$$
\begin{aligned}
\Psi \approx\ & \Theta \circ \\
& [1, 2, \ldots, n, (\Gamma n \rightarrow \mathrm{id}; \bar{\mathrm{F}}) \circ \Psi n \circ \Pi n] \circ \\
& \quad \ldots \\
& [1, 2, \ldots, i, (\Gamma i \rightarrow \mathrm{id}; \bar{\mathrm{F}}) \circ \Psi i \circ \Pi i] \circ \\
& \quad \ldots \\
& [1, (\Gamma 1 \rightarrow \mathrm{id}; \bar{\mathrm{F}}) \circ \Psi 1 \circ \Pi 1] \circ \\
& (\Omega \rightarrow [\mathrm{id}]; \bar{\mathrm{F}})
\end{aligned}
$$

Here, Θ is a FP function to construct a list of values of the output arguments for the predicate P. Θ constructs the list by selecting values from the list of input arguments of P and the value lists returned by the called (functions for) predicates $P1, P2, \ldots, Pn$. The FP function Πi $(0 < i \leqslant n)$ constructs a list of input argument values for function Ψi (Ψi represents predicate Pi in mode Mi). Like Θ, the function Πi uses the values of the input arguments of P and the outputs returned by (the functions for) predicates $P1, P2, \ldots, Pi - 1$. The FP function (guard) Γi verifies that the values of outputs returned by Ψi satisfy the unification requirements of the corresponding arguments in Pi. For example, if an output argument is specified as $[A \| B]$, Γ ensures that the value returned is a non-empty list. The FP function Ω checks that the input argument values can unify successfully with the corresponding input arguments of P in the head of the clause.

The construction of the functions Θ, Πi, Γi and Ω $(1 \leqslant i \leqslant n)$ can be based on the conventional methods of syntax-directed translation. We do not elaborate these rules here and leave it to the reader to device suitable schemes. As examples of the NFP definitions the definitions for the functions in the Front program are listed below.

$$
\begin{aligned}
\mathrm{Front} \equiv\ & 1 \circ [1 \circ 2] \circ [1, 2, \ \# \mathrm{Length} \circ [1 \circ 2, 1 \circ 1]] \circ \\
& (1, \ \# \mathrm{Unappend} \circ [2 \circ 1]] \circ [\mathrm{id}] \qquad (1)
\end{aligned}
$$

$$\text{Length} \approx [\ \bar{T}] \circ (\wedge \circ [= \circ [1, \bar{[}]], = \circ [2, \bar{0}]] \to [\text{id}];\ \bar{F}) \tag{2}$$

$$\text{Length} \approx [\ \bar{T} \circ [1, 2, \#\,\text{Length} \circ [t1 \circ 1 \circ, 1 \circ 2]] \circ [1, [- \circ [2 \circ 1, \bar{1}]]] \circ (\neq \circ [1, \bar{[}]] \to [\text{id}];\ \bar{F}) \tag{3}$$

$$\text{Unappend} \approx [\ \bar{[}], 1 \circ 1] \circ [\text{id}] \tag{4}$$

$$\text{Unappend} \approx [\text{Cons} \circ [1 \circ 1 \circ 1, 1 \circ 2], 2 \circ 2] \circ [1, \#\,\text{Unappend} \circ [t1 \circ 1 \circ 1]] \circ (\neq \circ [1, \bar{[}]] \to [\text{id}];\ \bar{F}) \tag{5}$$

The equivalence between a PROLOG clause and the associated NFP fragment can be justified based on the following observations:

(i) The non-deterministic choose operator, #, selects the appropriate defining fragments for the functions $\Psi 1, \ldots, \Psi n$.

(ii) The guards $\Gamma 1, \ldots, \Gamma n$ ensure that the values returned by functions $\Psi 1, \ldots, \Psi n$ satisfy the unification requirements on the corresponding output arguments of literals $P1, \ldots, Pn$.

(iii) The input arguments of the called functions are fashioned by functions $\Pi 1, \ldots, \Pi n$ on the basis of terms appearing in the input argument positions of the corresponding literals in the PROLOG clause. The called function ensures that its input arguments satisfy the necessary unification condition on the input arguments through guard Ω.

Using the rules of algebra given in Section 3, the NFP definitions of Front can be transformed into the following simplified forms:

$$\text{Front} \equiv (\ \bar{T} \circ \#\,\text{Length} \circ [2, 1] \to 2;\ \bar{F}) \circ [1, 1 \circ \#\,\text{Unappend} \circ [2]] \circ \tag{6}$$

$$\text{Length} \approx \wedge \circ [= \circ \bar{[}]], = \circ [2, \bar{0}]] \to [\ \bar{T}];\ \bar{F} \tag{7}$$

$$\text{Length} \approx \neq \circ [1, [\bar{[}]] \to (\ \bar{T} \circ \#\,\text{Length} \circ [t1 \circ 1, - \circ [2, \bar{1}]] \to [\ \bar{T}];\ \bar{F};\ \bar{F});\ \bar{F} \tag{8}$$

$$\text{Unappend} \approx [\ \bar{[}]], 1] \tag{9}$$

$$\text{Unappend} \approx \neq \circ [1, \bar{[}]] \to [\text{Cons} \circ [1 \circ 1, 1 \circ 2], 2 \circ 2] \circ [1, \#\,\text{Unappend} \circ [t1 \circ 1]];\ \bar{F} \tag{10}$$

5 CONSOLIDATING THE NFP DEFINITIONS

In this section we introduce two simple rules, CR1 and CR2, for consolidating the fragments defining a function into a single definition. A consolidated definition is acceptable if it is an extension of each of the fragments defining the function. That is, if for input I one of the fragments returns an output, the consolidated function must return the same output on input I. However, as it is not uncommon in PROLOG, if two or more

fragments of a function return different values for some input, the consolidated definition is free to return any one of these values. This condition is sometimes referred to as *'donot care'* non-determinism (e.g. [8]). Consolidation rule CR2 makes *'donot care'* choice of the fragments.

The CR1 rule, on the other hand, is designed to make *'donot know'* choice. Of the available choices only one is correct and will lead to some definite answer to the user query; the other choices lead to failures. However, a PROLOG interpreter may not know the correct choice when it is required to make the choice. The difficulty is overcome, in PROLOG, by searching over some or all of these choices. In our method, we use the algebra of NFP to transform the fragments into a form needed by CR1 rule. The rule then consolidates the fragments into a single definition incorporating the correct choice.

CR1: Let function $\circ V$ be defined by the the following NFP definitions:

$\Psi \approx C \to f; \bar{F}$ and,

$\Psi \approx \wedge \circ [\text{Not} \circ C, h] \to g; \bar{F}$.

These definitions can be replaced by a single definition:

$\Psi \approx C \to f; (h \to g;\ \bar{F})$

CR2: If the value returned by the function Ψ is not an input to any other function then CR1 can be somewhat relaxed. Thus,

$\Psi \approx r \to f; \bar{F}$ and

$\Psi \approx s \to g; \bar{F}$

can be replaced by

$\Psi \approx r \to f; (s \to g; \bar{F})$.

To illustrate the application of these rules to our example, we define two new NFP functions, Ok_length and Prefix, as follows:

$$\text{Ok\_length} \equiv \bar{T} \circ \# \text{Length} \tag{11}$$

$$\text{Prefix} \equiv 1 \circ \# \text{Unappend}. \tag{12}$$

Substituting the two NFP fragments (7) and (8) of Length into first of these definitions, simplifying and folding allows us to have following fragments for Ok_length:

$$\text{Ok\_length} \approx = \circ [1, \bar{[]}] \to = \circ [2, \bar{0}]; \bar{F} \tag{13}$$

$$\text{Ok\_length} \approx \neq \circ [1, \bar{[]}] \to \# \text{Ok\_length} \circ [t1 \circ 1, - \circ [2, \bar{1}]]; \bar{F}. \tag{14}$$

Using CR1 and some simplification, we get from (13) and (14):

$$\text{Ok\_length} \equiv = \circ [1, \bar{[]}] \to = \circ [2, \bar{0}]; \text{Ok\_length} \circ [t1 \circ 1, - \circ [2, \bar{1}]]. \tag{15}$$

Similarly, we can construct two NFP fragments for the Prefix function by unfolding (12) by (9) and (10);

$$\text{Prefix} \approx \bar{[]} \tag{16}$$

$$\text{Prefix} \approx \neq \circ [1, \bar{[]}] \rightarrow \text{Cons} \circ [1, \ \# \text{Prefix} \circ [t1]] \circ 1; \ \bar{\text{F}} \tag{17}$$

As a simple corollary to the above pair of definitions for Prefix, we conclude that Prefix is a total function.

$$\bar{\text{T}} \circ \# \text{Prefix} = \bar{\text{T}}.$$

On folding the definition of Front (6) using the functions Ok_length (11) and Prefix (12), we obtain

$$\text{Front} \equiv (\text{Ok\_length} \circ [2, \ 1] \rightarrow 2; \ \bar{\text{F}}) \circ [11, \ \# \text{Prefix} \circ [2]]. \tag{18}$$

Substituting the NFP fragments (16) and (17) of Prefix into (18), we obtain after some simplification

$$\text{Front} \approx \circ [1, \ \bar{0} \rightarrow \bar{[]}; \ \bar{\text{F}} \tag{19}$$

$$\begin{aligned} \text{Front} \approx \neq \circ [2, \bar{[]} \rightarrow (&\text{Ok\_length} \circ [2 \circ 2, \ 1] \rightarrow \text{Cons} \circ 2; \ \bar{\text{F}}) \circ \\ &[- \circ [1, \ \bar{1}], \ [1 \circ 2, \ \# \text{Prefix} \circ [t1] \circ 2]]; \ \bar{\text{F}} \\ \approx \neq \circ [2, \bar{[]}] \rightarrow \text{Cons} \circ [1 \circ 2, &\\ &(\text{Ok\_length} \circ [2, \ 1] \rightarrow 2; \ \bar{\text{F}}) \circ \\ &[1, \ \# \text{Prefix} \circ [2]] \circ [- \circ [1, \ \bar{1}], \ t1 \circ 2]]; \\ &\bar{\text{F}} \end{aligned} \tag{20}$$

Fragment (20) can be folded using (18) into the following form:

$$\text{Front} \approx \neq \circ [2, \bar{[]}] \rightarrow \text{Cons} \circ [1 \circ 2, \ \# \text{Front} \circ [- \circ [1, \ \bar{1}], \ t1 \circ 2]]; \ \bar{\text{F}}. \tag{21}$$

Using CR2 to consolidate the fragments (19) and (21) of Front, we get a recursive function for Front:

$$\begin{aligned} \text{Front} \equiv = \circ [1, \ \bar{0}] \rightarrow \bar{[]}; &\\ \neq \circ [2, \bar{[]}] \rightarrow \text{Cons} \circ [1 \circ 2, \ &\text{Front} \circ [- \circ [1, \ \bar{1}], \ t1 \circ 2]]; \ \bar{\text{F}}. \end{aligned} \tag{22}$$

PROLOG is not totally devoid of the control information. The order in which clauses are written does affect its execution behaviour. We may preserve some aspects of this behaviour in the functional program by maintaining the NFP fragment definitions for the functions in the order of their defining clauses. Indeed, the consolidation rules, as defined, favour the fragment defined earlier over the one defined later.

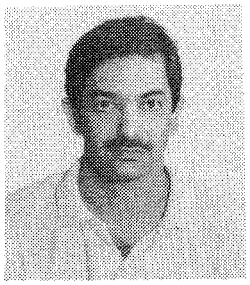
The methodology described in this paper has been used to translate several PROLOG programs into functional programs. The programs translated include: quick sort, bubble sort, insertion sort, prime numbers, and generation of spanning tree [9]. The methodology, however, requires that certain extra-logical features of PROLOG be handled externally, e.g. cuts and negative literals.

Efforts currently planned include investigation of methods for translating a wider class of PROLOG programs. Also, it is desired to have a computer-based system to perform the transformations. This will reduce the tedium of details that otherwise have to be handled manually.

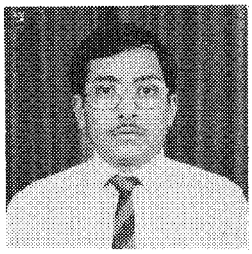
Acknowledgements. Part of the work reported here was done while the authors were at Indian Institute of Technology, Kanpur (India). We also gratefully acknowledge the suggestions made by the anonymous reviewers.

REFERENCES

[1] BACKUS, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM, 21, 1978, No. 8, pp. 613—641.
[2] BURSTALL, R. M.—DARLINGTON, J.: A transformation system for developing recursive programs. J. ACM 24, 1977, No. 1, pp. 44—67.
[3] CODD, E. F.: A relational model of data for large shared data banks. Comm. ACM 13, 1970, No. 66, pp. 377—387.
[4] DARLINGTON, J.—FIELD, A. J.—PULL, H.: The unification of functional and logic languages. In: [7], 1986, pp. 37—70.
[5] DEBRAY, S. K.—WARREN, D. S.: Detection and optimization of functional computation of Prolog. In: E. Shapiro (Ed.): Proc. of 3rd. Int'l. Conf. on Logic Programming, London. LNCS 225, Berlin, Springer-Verlag 1986, pp. 490—504.
[6] DEBRAY, S. K.—WARREN, D. S.: Automatic mode inference for Prolog programs. Proc. 1986 Symp. Logic Programming, IEEE, Sept. 1986, pp. 78—88.
[7] DEGROOT, D.—LINDSROM, G.: Logic programming: functions, relations, and equations. Prentice-Hall 1986, 533 pp.
[8] GREGORY, S.: Parallel logic programming in PARLOG. Addison-Wesley 1987, 217 pp.
[9] JAIN A.: On transformation of logic programs to functional programs, M. Tech. thesis, Dept. of Computer Science and Engineering, Indian Institute of Technology, Kanpur, 1987.
[10] MELLISH, C. S.: The automatic generation of mode declarations for Prolog programs. DAI Res. Paper 163, Dept. of AI, Univ. of Edinburgh, Aug. 1981.
[11] REDDY, U. S.: On the relationship between logic and functional languages. In: [7], 1986, pp. 3—36.



Vishv Mohan Malhotra graduated from Birla Institute of Technology and Science, Pilani (India) in electronics engineering. He then did his masters in electrical engineering and doctoral degree in computer science at Indian Institute of Technology, Kanpur. He was employed as a system analyst with Operations Research Group, Baroda during 1976—1977. Since 1981 he has been with Indian Institute of Technology, Kanpur where he is currently an assistant professor in the Department of Computer Science and Engineering. At present he is on leave from the Institute. His research interests include programming languages.



Anurag Jain graduated in electrical engineering from Indian Institute of Technology, Kanpur in 1985. He then worked for about a year with computer manufacturers; first with Hindustan Computers Limited, Hardwar and then with Uptron Digital Systems Limited, Lucknow. In 1986 he registered for his masters program in Computer Science, again at Indian Institute of Technology, Kanpur. He is currently a technical consultant with Citibank Overseas Software Limited, Bombay. His interests include the development of software systems for financial and banking applications.