Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 17-32

FUNCTIONS IN FULL PROLOG

T. Ásványi (Budapest, Hungary)

Abstract. A functional notation for Prolog predicates, which is well known in the Prolog community, but its implications are not fully explored, is improved. A consistent notation, called fnProlog, is developed by the author, giving solutions to problems unhandled before: 1. Calls to built-in (arithmetic) and user-defined functions can be intermixed in function expressions without limitations. 2. The notation is more Lisp-like, and easier to read by introducing equivalents of quoting and back-quoting mechanisms. 3. FnProlog is actually an extension of SICStus Prolog 2.1 #9. It is adapted to the module system of SICStus. 4. Function invocations inside calls to meta-predicates work correctly. 5. Function invocations can be used in DCG rules without limitations. 6. The notion of meta-function and of meta-DCG rule is introduced. The implied problems are solved. 7. Dynamic updating of functions is possible.

1. Introduction

Several systems uniting functional and logic programming have been developed [1]. Some of them have separate logic and functional components communicating through an interface. They have the full power of the constituent languages, (for example, Lisp+Prolog), in the components [9]. The interface is normally complicated and slow. Warren (1974, Problem 85 in [7]) suggested an optional functional notation for Prolog predicates. Phil Vasey and others [4] improved it. These "functions" keep the expressive power of backtracking and incomplete data structures added to the expressive power of pure Lisp [10]. Inside a Prolog goal, a function invocation can be in the syntactic position of a term. It is syntactically a Prolog term with some special prefix.

Function invocations use pattern matching for parameter passing. The functions (that is predicates in functional notation) consist of function clauses, which are expanded to normal Prolog clauses during compilation. Similarly, the goals containing function calls are expanded to normal Prolog goals. Therefore function calls involve no interface cost. The approach is the one followed in *Definite Clause Grammars* (DCG) and *Object-Oriented* extensions [2,4]: the function clauses and the other clauses containing function calls are expanded into normal Prolog clauses using the standard hook-predicate `term_expansion/2`. The body of a function clause consists of a Prolog goal sequence. Therefore Prolog calls (from functions) need no interface. Functions involve no runtime overhead.

Our functional notation adopts the concepts of Phil Vasey's notation enumerated above, but we introduce the notion of function expression, which is a generalization of arithmetic expression. Calls to user-defined functions and calls to built-in (arithmetic) functions can be intermixed in function expressions without limitations. A function expression is syntactically a Prolog term with a question mark as a prefix. `(:- op(650,fy,?).)` (See 2.1-2.2 and 6.)

We adopt the quoting and back-quoting notations of Lisp, so that we can write data structures (normal Prolog terms) into the arguments of function calls, and function calls again into the arguments of those data structures. (See 2.2.)

As our functions are backtrackable, we introduce a special built-in function called `findall` for collecting all the results of a function expression into a Prolog list. (See 2.4.)

We solve the consistency problems arising from the possibility of using function calls in the goals written into the meta-arguments of predicate calls (see Chapter 3). We are aware of the fact that functions and DCG rules may also have meta-arguments interacting with function calls specially. The implied problems are discussed in 3.1 and 3.2.

We introduce some new built-in predicates for dynamic updating of functions. These are `assertf/1, assertaf/1, assertzf/1, retractf/1, retractallf/1, ':='/2`. The last predicate changes the value of a given dynamic function with given arguments. The others are the counterparts of the corresponding predicates for dynamic updating of normal Prolog predicates (see Chapter 4).

We compare our functional notation to that of APPLOG and Prolog++ 1.0 in Chapters 5 and 6. As far as we know, these two software systems have the most sophisticated functional notation among the predecessors of our fnProlog [1,4,6,7].

FnProlog 2.3 inherits all the concepts of SICStus Prolog 2.1 #9 [2] and it does not change them. A SICStus Prolog module (called fnProlog) has been written for expanding clauses and goals containing functional elements to normal Prolog clauses and goals respectively.

2. Basic concepts

According to [8] pure Lisp can be considered as a simplification of pure Prolog, and such Lisp programs are not even faster than their Prolog equivalents, if cuts are allowed. On the other hand, Lisp functions are often more concise (due to fewer auxiliary variables) and easier to understand (due to clear data flow) than their Prolog equivalents.

If one wants to add functional notation to Prolog, one has to use the same notion of data in both components of the system, if one does not want to have run-time interface costs between them. For example, when data are passed from Prolog to Lisp, recursive dereferencing of terms is needed, as seen e.g. in Poplog [9]. Therefore we use Prolog terms for representing data. We do keep the concepts of traditional Prolog in our functional notation as far as possible, so that the Prolog programmer does not feel strange with the new ideas. We show our basic concepts through some examples in the next paragraph.

2.1. Asymmetric relations

Let us suppose that there are facts about family relationships, for example:

```
father(set,adam).
mother(set,eve).
```

It is not clear, how to read this: is Set the father of Adam or is Adam the father of Set? Functional notation is more clear:

```
father(set) = adam.
mother(set) = eve.
```

These clauses are equivalent to the previous ones, and will be expanded to them during compilation, but the readability has been increased: relations are often directed, and this fact can be expressed by functional notation.

It is desirable that a function could be defined by another function:

```
parent(X) = ?father(X).
parent(X) = ?mother(X).
```

```
grandparent(X) = ?parent(parent(X)).
```

The function expressions are prefixed by a question mark operator, (`:- op(650,fy,?).`), that is, function `parent/2` expresses that one's parents are his father and mother. (The arity of a function is the arity of its head plus one, that is, its arity is the arity of the predicate it is expanded into.) The question mark shows that a function expression defines a value (or values) of the function instead of a simple Prolog term, that is:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

are the Prolog clauses equivalent to the above function clauses.

Calls to user-defined and built-in functions are handled in a uniform way, and they can be mixed freely, unlike in previous systems. (Compare 2.1 and 2.2 with 6 and [4,7].) Let us consider another example:

```
fib(N) = ?fib(N-1)+fib(N-2) :- N>1.
fib(1) = 1.  fib(0) = 1.
```

this is expanded to:

```
fib(N,FN) :- N>1,
    N1 is N-1, fib(N1,FN1),
    N2 is N-2, fib(N2,FN2),
    FN is FN1+FN2.
fib(1,1).  fib(0,1).
```

For the exact syntax of function clauses, and for the detailed rules of their expansion into normal Prolog clauses, see 2.3.

2.2. Activating functions

Most often we write function calls with a '?' prefix into the arguments of Prolog goals, because it is the most convenient way to activate functions:

```
write_fib(N) :-
    writeseq( [ 'The ', N, '.  fibonacci number is:  ',
           ?fib(N), '.'  ]  ), nl.
```

Module `fnProlog` expands this clause into:

```
write_fib(N):-
    fib(N,FN),
    writeseq( ['The ',N,'.  fibonacci number is:  ',FN,'.']  ),
```

```
nl.
```

If some subterms of goals are function calls, first these subterms are evaluated to temporary variables by goals expanded from those subterms. Then the goal is called with the temporary variables. If a function call has the arity N, it is expanded into a Prolog call of arity N+1, where a new last argument is added to the call. This new argument contains the appropriate temporary variable. If a function call is an arithmetic expression, it is expanded into the appropriate `is/2` call.

We support meta-calls to functions, too. A variable with a question mark prefix, e.g. `?Var`, is interpreted as a meta-call to a function-expression and the Prolog goal `Temp eq Var` is generated to deposit its value into `Temp`. The predicate `eq/2` is an auxiliary procedure to perform expansion at run-time. If `Var` is actually a variable or a number, `Temp eq Var` is expanded to `Temp = Var`. Otherwise it is expanded as `Temp = ?Var` would be expanded with the actual value of `Var`. When this expansion has been finished, its result is executed, for example:

`write_result(X) :- write(?X).` is expanded to

```
write_result(X) :- Temp eq X, write(Temp).
```

Let us suppose that the call `write_result(fib(5))` is invoked. According to our rules, `Temp eq fib(5)` would be expanded to `fib(5,TempX), Temp=TempX`. Due to some optimizations it is expanded to `fib(5,Temp)`. If `write_result(3*N+2)` is invoked, `Temp eq 3*N+2` is expanded to `Temp is 3*N+2` as it is natural.

Arguments of function invocations are interpreted similarly to those in Lisp. Numbers and variables are not touched by the expansion of the call. Atoms and compounds are normally expanded as inner function calls. Because these arguments of function calls are again considered function calls, the function calls form *function expressions*, giving a natural *generalization of arithmetic expressions*.

The goals produced from the arguments of a function invocation precede the goal expanded from that function invocation, for example:

The goal `write(?sort(randomlist(Length)))` is expanded into the goal sequence `randomlist(Length,List), sort(List,Sorted), write(Sorted)`. (`List and Sorted` are new temporary variables.)

Marking with asterisk is introduced so that we can write atoms and compound terms as data into the arguments of function calls. A term marked with an asterisk is of the form `*Term`. `(:-op(650,fy,*).)`. If `*Term` is an argument of a function invocation, this marked argument is expanded as `Term` would be expanded in the position of an argument of a predicate call. This

means that `*Term` is simply replaced by `Term` in the actual argument, except in the case when subterms of the form `?FuncCall` are encountered.

For example, the goal `write(?ins(X,*t(?fib(8),void,T)))` is expanded to `fib(8,Root), ins(X,t(Root,void,T),NewTree), write(NewTree)`.

The atom [] and the functors `'.'/2` and `','/2` are implicitly marked with asterisks. This means that normal lists and round lists are considered to be Prolog terms, even in the syntactic position of (an argument of) a function invocation. Their elements may be function calls anyway. This rule is adopted for convenience in the case of normal lists, and to reduce the number of coding mistakes in the case of round lists, for example:

The goal `write(?reverse((2,?cos(fib(4)-X+1))))` is expanded into
`fib(4,F4), C3 is cos(F4-X+1), reverse((2,C3),R), write(R).`

If we wrote an asterisk before the round list omitting the necessary blank character, it would be interpreted as a function call.

A term marked with a double asterisk is of the form `**Term`. (`:- op(650,fy,**).`). If a subterm of a function or predicate invocation is marked with a double asterisk, then this subterm (`**Term`) is simply replaced by `Term` at the actual position. This rule applies even in the case when subterms of `Term` matching the form `?FuncCall` are in `Term` (see `assertf/1` in 4.)

Note that the asterisk notation corresponds to the "back-quote", and the double-asterisk to the "quote" notation in Lisp. (Quotes are atom delimiters in Prolog and back-quotes are string delimiters in some implementations [2,5], that is why we do not use the Lisp notation.)

Unlike in previous systems [4,7], one can write goal sequences containing function calls at the Prolog prompt, for example:

```
?- ?sort([3,2,4,1]) = ?append([1,2],[3,4]).
```

2.3. Function clauses

A function clause is written in the form:

`Head = Term.` % `Head` is a Prolog atom or a compound. `Term` is a Prolog term

or

`Head = Term :- Body.` % `Body` is equivalent to a normal clause body.

During compilation every function clause is expanded to a normal Prolog clause. Therefore the arity of a function is the arity of its `Head + 1:`. While expanding the `Head`, a new last argument (generated during the expansion of

Term) is added to it. If Term contains subterms with question mark prefixes, they are expanded as function invocations and the goal sequence generated will be appended to the expanded Body.

2.3.1. Examples of function expansions:

```
element_of([X|_]) = X.    % Function element_of/2
element_of([_|L]) = ?element_of(L).

element_of([X|_],X).      % is expanded to this tail recursive predicate.
element_of([_|L],X) :- element_of(L,X).

append([ ],L) = L.        % Function append/3
append([H|T],L) = [H|?append(T,L)].

append([ ],L,L).          % is expanded to this tail recursive predicate.
append([H|T],L,[H|TL]) :- append(T,L,TL).

ins_sort_tree(void,X) = t(X,void,void).  % This function is
                                           expanded
ins_sort_tree(t(Root,Left,Right),X) =
    t(Root,?ins_sort_tree(Left,X),Right):- X@=<Root, !.
ins_sort_tree(t(Root,Left,Right),X) =
    t(Root,Left,?ins_sort_tree(Right,X)).

ins_sort_tree(void,X,t(X,void,void)).  % into this tail rec. pred.
ins_sort_tree(t(Root,Left,Right),X,t(Root,LeftX,Right) :-
        X @=< Root, !, ins_sort_tree(Left,X,LeftX).
ins_sort_tree(t(Root,Left,Right),X,t(Root,Left,RightX) :-
        ins_sort_tree(Right,X,RightX).
```

2.4. Functions in Lisp and fnProlog

It is clear now that fnProlog functions may be indeterministic and they might return nonground terms (variables or terms containing variables). These properties are different from the properties of normal (for example Lisp) functions. When Prolog is extended by functional notation, it seems to be reasonable to retain the expressive power of indeterminism and incomplete data structures.

Let us suppose that we have the following database:

```
parent(a)=b.  parent(a)=c.  parent(b)=d.  parent(b)=e.  ...
```

The functions ancestor and ancestors can be defined in an elegant way now:

```
ancestor(X)= ?parent(X).
ancestor(X)= ?ancestor(parent(X)).
ancestors(X)= ?findall ancestor(X).  % :-op(650,fy,findall).
```

`findall/2` is a special built-in function in fnProlog. It may have a function expression in its single argument. It returns all the solutions of that function expression.

The call `?ancestors(*a)` returns the list of all the ancestors of `'a'` with multiplicity.

If backtrackable functions were not allowed, one would have to define the function `ancestors/2` in Lisp style, for example (adding some cuts, it works in fnProlog):

```
parents(a)=[b,c].  parents(b)=[d,e]...  % Analogous to a property list.
parents(_)=[ ].                         % Otherwise
ancestors(X) = ?ancestors(parents(X),[ ]).
ancestors([P|Ps],Ancestors) =
    ?ancestors(append(parents(P),Ps),[P|Ancestors]).
ancestors([ ],Ancestors) = Ancestors.
```

This definition is more complicated and cannot be used in such a flexible way: You have to write another program to decide, if one is ancestor of another.

The concatenation of d-lists can be defined in the following form:

```
conc(L1-L2,L2-Z) = L1-Z.
```

Without incomplete data structures, one has to use destructive operations for effective concatenation of lists [3,6,8,10].

Functions are even reversible:

```
prefix(L) = L1 :- ?append(L1,_) = L.
```

This last use is not encouraged. (Functions suggest a data flow.) The following definition is more elegant:

```
prefix(_)=[ ].
prefix([H|T]) = [H|?prefix(T)].
```

2.5. Functions and modules

Functions can be exported from and imported by modules, because they are expanded to predicates. Their invocations can even be prefixed by the module name overriding the standard visibility rules, just like in the case of predicates [2]. Maintaining that functions, modules and meta-predicates work together in a consistent way, was the most difficult task while developing the concepts and the implementation of fnProlog, but the programming difficulties solved are out of the scope of this article.

3. Meta-predicates

When the module system of SICStus Prolog is used, some special kinds of arguments (files, predicates, clauses and goal sequences) of predicate calls are to be handled specially during compilation. Therefore so-called meta-predicate declarations are needed. For example:

```
all_of(P) :- P, fail.
all_of(_).
```

and

```
for_all(P,Q) :- \+ ( P, \+Q).
```

are meta-predicates, because their arguments are supposed to be goals. Therefore they need meta-declaration as it is specified in [2]:

```
:- meta_predicate all_of(1), for_all(1,2).
```

(In a meta-declaration a colon or an integer indicates that the corresponding argument is to be handled specially.)

When one uses fnProlog, a new problem is raised by meta-predicates. Let us suppose, that an argument of a meta-predicate invocation is a goal, for example:

`all_of( process(?element_of(L)) )` would be expanded to

`element_of(L,X), all_of(process(X))`, according to our rules, instead of

`all_of( ( element_of(L,X), process(X) ) )`, which is desirable.

To solve this problem, we distinguish arguments of predicates expecting goal sequences (we will call them meta-arguments), arguments expecting other kinds of special parameters (predicates, clauses or files), and "normal" arguments. Meta-arguments are denoted by integers between 1 and 100 (practically

by their sequence number), and other kinds of special parameters by colons or integers out of the range 1-100. When a meta-argument of a meta-predicate call is encountered while expanding the call (for example, during compilation), the function calls inside it are not expanded textually before the calling of the meta-predicate. They are expanded inside the goal sequence of the meta-argument. For example, the following implicit declaration for the built-in `findall/3` is given:

```
:- meta_predicate findall(?,2,?).
siblings(A,Siblings) :-        % This clause is expanded into
        findall(X,?parent(A)= ?parent(X),Siblings).
siblings(A,Siblings) :-        % this one, as it is expected.
        findall(X,(parent(A,Z),parent(X,Z)),Siblings).
```

As far as we know, fnProlog is the first functional notation handling the functions together with built-in and user-defined meta-predicates in this consistent way.

3.1. Meta-functions

Let us have a look at predicate `siblings/2` above. It would be more natural to formalize it as a function:

```
siblings(A) = ?findall( X, ?parent(A) = ?parent(X) ).
```

This example shows that functions may have meta-arguments, for example the second argument of the function call to `findall/3` here. (It also shows that even traditional predicates are sometimes called using the syntax of function invocations, if it is sensible.)

A meta-function is one to be expanded into a meta-predicate. When we write calls to meta-functions into function expressions, their meta-arguments are expanded as goal sequences, because these arguments normally contain goals.

Meta-functions are introduced so that functions parameterized by goals containing other function invocations work in the natural way. That is function `siblings/2` above will be expanded into the expanded version of predicate `siblings/2` above.

We write meta-predicate declarations even for meta-functions, because functions are just notational variants of predicates:

```
:- meta_predicate seconds(1,-).
seconds(Call) = ?Milliseconds/1000 :-
```

```
statistics(runtime,_), Call,
statistics(runtime,[_,Milliseconds]).
```

3.2. Meta-DCG rules

A meta-DCG rule is one to be expanded to a meta-predicate clause. For example:

```
:- meta_predicate dcg(1,-,?,?), notion(1,-,?,?).
dcg(Cond,N) --> Cond, notion(Cond,N).
```

The problem that function invocations inside `Cond` are to be expanded inside the goal sequence of the corresponding argument of `dcg/4` is solved in fnProlog.

4. Destructive operations and side effects

Compared to Prolog, side effects and destructive operations can be implemented effectively in Lisp. Their performance is similar to that of the equivalent operations in procedural languages. For example, the cost of `setq` is comparable to the cost of pointer assignment.

In Prolog (even if using functional notation) these effects are achieved mainly through the `assert/retract` facility, because it corresponds to the relation-based approach and non-backtrackable assignment statement is implemented with copying the whole data structure. (The data structures built incrementally may be corrupted by backtracking.)

In the current version of fnProlog, one may assert and retract dynamic function clauses. One may update the values of dynamic functions with non-backtrackable assignment statement. After all, assignment depends on the `assert/retract` facility:

- `assertf/1` expands its argument to a clause not containing functional notation, and asserts the resulting clause. Its argument can be a clause, a function clause or a DCG rule. `assertaf/1` and `assertzf/1` work similarly, but they call `asserta/1` and `assertz/1` after the expansion, respectively. The argument of `assertf/1` may need a double-asterisk prefix, so that subterms of the form `?FuncCall` are expanded correctly. For example:

```
... , assertf( ** (append([H|T],L)=[H|?append(T,L)]) ), ...
```

- `retractf( f(T1, ... ,Tn) )` retracts the first clause of (the dynamic) predicate `f/(n+1)`, whose head matches `f(T1, ... ,Tn,_)`. It is indeterministic like `retract/1`. It does not depend on the body of the clause, unlike `retract/1`, because the body of the clause has probably been expanded by `assertf/1` or during compilation.
- `retractallf( f(T1, ... ,Tn) )` retracts all of the clauses of (the dynamic) predicate `f/(n+1)`, whose head matches `f(T1, ... ,Tn,_)`.
- `f(T1, ... ,Tn) := Term` expands `Term` and evaluates the resulted goals (for example, to `Result`), calls `retractall( f(T1, ... ,Tn,_) )`, and asserts the function clause: `f(T1, ... ,Tn) = Result`. It is intended to be used mainly from the Prolog prompt, so that one can change some parameters of a program conveniently. For example the following query runs successfully:

```
?- c:=fib(5), d:=3, a(2):= ?d+(?c),
    ?c=fib(5), ?d=3, ?a(2)=11.
```

(Note that the predicates in this chapter do not have meta-arguments. Therefore those of `T1, T2,..., Tn` containing function calls are expanded according to the basic rules.)

Dynamic updating of functions is a new feature of fnProlog compared to other functional extensions [1,4,6,7]. It is parallel to dynamic updating of predicates.

5. Comparing fnProlog to Applog

Applog is an earlier attempt to extend Prolog with a (Lisp-like) functional component, which has a "practically invisible" interface [1]. This extension written in C-Prolog adopts the term and variable notion of Prolog. It uses pattern matching for parameter passing. It contains almost all the important pure functional concepts of Lisp. Calls to built-in and user-defined functions can be mixed freely in function expressions, handling them in a uniform way. In spite of the fact, that its function definitions are syntactically Prolog facts, they are very similar to Lisp [10] definitions:

```
def(append,lambda([L1,L2],      % Unfortunately, this definition of ap-

    if( eq(L1,[ ]), L2,         % pending lists is not tail recursive.
                                    (See 2.3.1)
        cons(car(L1),append(cdr(L1),L2)) ))).
```

Applog includes the backtracking possibility even in the functional component: indeterministic calls to Prolog relations or goals are possible, and backtracking is activated, if some subsequent function evaluation fails. In such way it incorporates the expressive power of backtracking into a Lisp-like programming tool. Programming of side effects depends on the `assert/retract` possibility of the underlying Prolog system. Functions can be invoked only through the predicate `eval/2` (activating an interpreter), for example:

```
eval(append(L1,L2),L1L2).
```

(A function call cannot be written into an argument of a Prolog goal, except of this special case.)

The function definitions are interpreted in C-Prolog. It means that there are no problems with meta-predicates. This fact, however, implies three new problems:

- The functions defined work considerably slower than the equivalent Prolog programs.
- The back-quoting possibility is missing, because it is ineffective to scan big data structures for subterms representing function calls. (Therefore there is no notation for subterms representing function calls.)
- Even if the actual parameter of a function call is textually a variable, when this call is executed it is tried to be interpreted as a function call. If we intend to match a compound or an atom to this actual parameter, it must be quoted, breaking the tradition of Lisp.

6. Comparing fnProlog to Lpa Prolog++ 1.0

Lpa Prolog++ 1.0 [4] is a powerful object-oriented extension of Prolog. Functions can be defined and used only inside objects (or instances of objects). Therefore functions cannot be used independently from objects. Otherwise, its notion of function is very similar to that of fnProlog, as it has been described in the Introduction of this paper. Let us have a look at the following example:

```
open_object mix.

has([X|_]) = X.          % Equivalent to the fnProlog definition
has([_|L]) = self::has(L).      % of element_of/2 in 2.3.1.

write_list(L) :- for_all( write(self::has(L)), nl ).
% for_all/2 is defined in 3.
```

```
writelist(L) :- forall( write(self::has(L)), nl ).
% forall/2 is built-in.
% fib(N) is self::fib(N-1)+self::fib(N-2) :- N>1.
% N-1 and N-2 above are not evaluated !
fib(N) is self::fib(N1)+self::fib(N2) :-
        N>1, N1 is N-1, N2 is N-2.
fib(1) is 1.  fib(0) is 1.
close_object mix.
```

The appropriate object name and the double colon operator are the necessary prefixes to function calls. Therefore every single function invocation is marked instead of marking the function expressions and using some back-quoting mechanism, when necessary. There is no possibility to evaluate arithmetic expressions in the actual parameters of function calls. Therefore function `fib/2` is to be written in the form given above. As the example shows, this notation is not as flexible as the notations introducing function expressions.

There is no notation for quoting. There are no meta-functions nor meta-DCG rules. There is no support for dynamic updating of functions, except that attributes (which can be considered functions with no parameters and no body), can be changed by assignment statement (depending on the `assert/retract` facility).

Function calls in the meta-arguments of built-in meta-predicates are handled correctly. For example, the call `?- mix<-writelist([1,2,3]).` writes out correctly `1`, `2` and `3` on subsequent lines. (`forall/2` is a built-in meta-predicate "equivalent to" `for_all/2` defined in 3.)

On the other hand, there is no way to declare user-defined meta-predicates. Therefore the call: `?- mix<-write_list([1,2,3]).` outputs only the first element of the list. Similarly, the call `?-mix<-writelist([]).` succeeds, but the call `?- mix<-write_list([]).` fails.

The problems with arithmetic expressions and quoting are to be solved in the new release of Prolog++, according to the author's correspondence with LPA.

7. Conclusions

Knowing the solution of Problem 85 in [7], one may think that the task of introducing an optional functional notation for Prolog predicates is trivial. Analysed carefully, this notation turns out to have interaction with many other concepts of Prolog. Facing them, a new functional notation has been developed and implemented.

Instead of prefixing each single function call with its own question mark, whole function expressions are marked and quoting or back-quoting is used, where necessary. In this way we avoid the proliferation of question marks in the function expressions and the notion of function expression is a generalization of the arithmetic one, following the tradition of Lisp.

Functions work together with modules and DCG rules as it is expected.

It is recognized that function calls inside the meta-arguments of meta-predicate invocations are to be expanded inside the corresponding arguments in the case of user-defined meta-predicates as well as in the case of built-ins. Meta-functions and meta-DCG rules are introduced, too, to make sure that the function calls are expanded locally to the meta-arguments of the function and DCG rule invocations.

The dynamic updating of clauses containing functional elements becomes necessary. Therefore it is supported by some new predicates corresponding to their classical counterparts.

FnProlog is implemented entirely in SICStus Prolog 2.1 #9, which is one of the most popular LP systems. It can be easily adapted to other platforms, too, because SICStus follows the standards introduced by Edinburgh and Quintus Prolog.

Similarly to the DCG notation, fnProlog does not want to gain new application areas for Prolog. Predicates expressing directed or asymmetric relations can be coded often elegantly using this improved, carefully examined notation, increasing the clearness and the self-documenting feature of Prolog programs.

Acknowledgements. I am indebted

- to David H. D. Warren and to the authors of Lpa Prolog++ 1.0 [4], because my concepts on functional notation for Prolog are based on their ideas,
- to Anna Bagyinszki-Orosz and Péter Szeredi for their critical comments,
- to László Varga and János Demetrovics for support in purchasing the necessary software systems,

- to my wife for her patience.

References

[1] **DeGroot D. and Lindstrom G.,** *Logic Programming: Functions, Relations and Equations,* Prentice Hall, 1986.

[2] **Carlsson M., Widén J. et al.,** *SICStus Prolog 2.1 #9 User's Manual + Library Manual,* Swedish Institute of Computer Science, POB 1263, S-164 28 Kista, Sweden, 1994.

[3] **Sterling L. and Shapiro E.,** *The Art of Prolog (second edition),* The MIT Press, London, England, 1994.

[4] **Vasey P., Spencer C., Westwood D. and Westwood A.,** *Lpa Prolog++ 1.0 Programming Reference Manual,* Logic Programming Associates Ltd., Studio 4, The Royal Victoria Patriotic Building, Trinity Road, London SW18 3SX, England, 1994.

[5] **Westwood D.,** *Lpa Prolog 2.6 Programming Guide,* Logic Programming Associates Ltd., Studio 4, The Royal Victoria Patriotic Building, Trinity Road, London SW18 3SX, England, 1994.

[6] **Amble T.,** *Logic Programming and Knowledge Engineering,* Addison-Wesley, 1987.

[7] **Coelho H., Cotta J.C. and Pereira L.M.,** *How to Solve It with Prolog,* Laboratória Nacional de Engenharia Civil, Lisboa, 1980.

[8] **Warren D.H.D., Pereira L.M. and Pereira F.,** Prolog - The language and its implementation compared with Lisp, *Proc. ACM Symp. on AI and Programming Languages, Rochester, N.Y., 1977., SIGPLAN Notices,* **12** (8) (1977), 109-115.

[9] *PopLog User Guide,* Integral Solutions Ltd., University of Sussex, England, 1994.

[10] **Steele G.L.,** *Common Lisp: The Language, 2nd edition [CLtL2],* Digital Press, 1990.

T. Ásványi
Department of General Computer Science
Eötvös Loránd University
VIII. Múzeum krt. 6-8.
H-1088 Budapest, Hungary
asvanyi@ludens.elte.hu