User's Functions in Standard Prolog

Tibor Ásványi

Dept. of Comp. Sci., Eötvös Loránd Univ.
Budapest, Pázmány Péter sétány 1/c, H-1117
asvanyi@inf.elte.hu

Abstract. We integrate user-defined functions with semantic equations into standard Prolog. In order to interface them with Prolog, we introduce a single operator. If a predicate invocation is prefixed by this operator, its parameters are expressions evaluated by the standard arithmetic of Prolog, and by our equations using innermost basic narrowing. The extended language is called PLN 5. (PLN = ProLog with Narrowing).
PLN 5 conforms to the ISO standard of Prolog without compromise. It is a simple but flexible, conservative functional extension of ISO Prolog. It does not introduce runtime overhead, and it avoids error-prone constructs. It adds the expressiveness of functions to the notational power of Prolog.
PLN 5 is implemented in a SICStus Prolog module, and it is fully integrated with this host language. The implementation depends on the term- and goal-expansion capabilities of SICStus.

1 Introduction

Our aim is to extend the Prolog arithmetic to user-defined and non-arithmetic evaluation. In the extended language called *PLN 5*, the user can define his own functions. Then he can freely mix the calls to his functions with the calls to the predefined functions (of Prolog and PLN) in his *pln expressions*.

Our functions are defined by equations, stating that syntactically different (i.e., non-unifiable) terms are semantically equal [2, 5]. Although this is a usual solution in functional-logic programming (FLP), we feel that adding equations to standard Prolog may work against the structured programming style: New functions may turn terms representing data structures to terms representing functional expressions, and this may unexpectedly change the meaning of the old code, especially, the run of the goals parameterized by those terms. After all, our aim is only to *evaluate* expressions. Therefore serious restrictions are imposed on the use of equations, according to the following constraints:

(C1) The extension introduced must not alter the underlying language [1, 6].
(C2) If a PLN program is given, then new pln functions must not unexpectedly change the meaning of the old code.
(C3) The new constructs must not cause runtime overhead compared to standard Prolog programs.

In order to satisfy (C1), we introduce *pln goals*. They are predicate invocations (or expandable goals [6]) prefixed by a `?` operator: `op(900,fy,?)`. Their parameters are *pln expressions*. When a pln goal is performed, first the parameters of the prefixed call are evaluated. Then the pln expressions are replaced by their values, and the resulting Prolog goal (without its `?` prefix) is performed. (For example, the Prolog goal `write(f(N)+1)` prints the *term* `f(N)+1`, but the pln goal `?write(f(N)+1)` prints the *value* of `f(N)+1`.)
(C2) is intended to support structured programming. This point will explain the need of introducing the *pln goals* used to force the evaluation of the pln expressions (sect. 2.1), and the need of *constructor definitions* (sect. 3).
(C3) means that the most of our constructs are handled during compilation. A pln goal can be considered a call to the meta-predicate `'?'/1`, but our precompiler substitutes it with an appropriate goal sequence (sect. 4).

1.1 The History of PLN

Similar constraints were already adopted by the author in his first work [9], but they were not made explicit there. From [10] these requirements became in the focus. The first extension called *fnProlog* [9] used the old *Lisp-like* back-quoting construct [7] to include explicit data structures in evaluable expressions. In [10] the extension became *constructor-based*, and the new constructs were properly integrated with the module system of SICStus. In [11] the notation and the terminology was highly developed. In [12] we introduced exception handling. Whilst in [9] the compilation issues were in the centre, later the concepts and the semantics of the language became more and more independent from the implementation.

However, in the previous versions [9–12] functional expression evaluation was forced by the prefix operator `op(650,fy,?)` if the term of the form `?`*term* was (a subterm of) the actual parameter of a goal. This method was inherited from Prolog++ [4]. It caused some consistency problem: The functional expressions inside meta-goals[1] were not always handled. (This was partly known and declared referring to effectivity problems.) In addition, the resolution of the consistency problems raised by the interactions of evaluation forcing expression prefixes and meta-predicate[2] calls made the extension rather complex: This feature, especially the notions of *meta-* and *template-arguments* deserved the most criticism on the language.

Provided that the complete actual parameters of the predicate invocations can only be forced to be evaluated, these consistency problems disappear. How-

[1] A *meta-goal* is a member of a goal sequence, but in the source code of the program it is usually a variable [1, 6]. When it is called during the run of the program, it must be instantiated to an appropriate goal (sequence). It is (pre)compiled and then the resulting sequence of predicate invocations is evaluated.

[2] A *meta-predicate* usually has some meta-goals in the bodies of its clauses, and these meta-goals are among its formal parameters. When the meta-predicate is invoked, the appropriate actual parameters must be goals. For example, `findall/3` and `goal/1` are predefined meta-predicates of Prolog [1, 6].

ever, the terms with functor `'?'/1` are still specially handled, and this conflicts with the ISO standard. In addition, the evaluation of the meta-goals needs a significant amount of additional run-time checks even if functional expressions are not used in them.

In PLN 5 [13] all these problematic concepts are replaced by the author's simple notion of *pln goals*: Just two new built-in predicates, `'?'/1` (sect. 1) and `constructor/2` (sect. 3) are introduced (and three new operators). Therefore this extension is allowed by the standard [1]. It is easy to avoid the use of predicate `'?'/1` in the meta-goals, because the pln functions can be invoked with the syntax of predicate calls (sect. 4). Then (compared to Prolog) there is no run-time overhead on the evaluation of (meta-)goals.

The notion of *pln goals* is the author's reply to the popular demand for introducing user defined functions into standard Prolog "with extensions to the functionality of `is/2`" [3].

Now we have a completely new notion of *meta-expressions*. It is more flexible and allows more effective implementation than the old one.

The implementation techniques were improved in general. As a result, the current implementation written in SICStus Prolog 3.10 is simpler and faster, it generates better Prolog code and it works together smoothly with other extensions of SICStus which use the *term* and *goal expansion* facilities [6].

1.2 The Organisation of This Paper

In the next section we introduce our *pln functions* and *pln expressions*. The concept of *narrowing* comes from [2], but we introduce it as a generalization of arithmetic evaluation based on [4], according to the demand stated in [3]. In the third section we introduce *constructors* [2]. The need of *constructor definitions* in a conservative functional Prolog extension is recognised and argued. In the fourth section the most important features of the actual implementation are dicussed. In the fifth section we discuss some built-in functions of PLN 5 [13].

Instead of a formal approach, the author prefers to give a taste of his Prolog extension, and to explain, why, and how the more important new notions and constructs were introduced.

2 Basic Ideas of Expression Evaluation

Following the idea of Phil Vasey [4], we generalize Prolog [6] arithmetic with semantic equations (sect. 2.1), which are applied only if forced, and only in left-to-right direction, using innermost basic narrowing[3] [2].

[3] *Narrowing* means the following: When an equation is applied, its *head* (sect. 2.1) is unified with the subexpression to be evaluated. During the unification the variables of the expression to be evaluated may be instantiated. Then the search tree of the corresponding goal sequence grows narrow. *Basic* narrowing means that the

2.1 PLN Functions and Expressions

Following [2, 4] a *pln function* is a sequence of *pln clauses*, and a pln clause is a semantic equation, that is

- a pln rule of the form: *head* `=` *expression* `:-` *condition.*
- or a pln fact of the form: *head* `=` *expression.*

Syntactically *head* is a compound term, *expression* is a term representing a *pln expression*, and *condition* is a goal. The corresponding function is identified by the *name* and *arity* of the principal functor of *head*, and we will refer to it in the form *name//arity*. For example, the pln function `fib//1` (calculating Fibonacci numbers) may be defined as follows:

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

According to our first constraint (C1) the goal `X=fib(2)` is only a syntactic equation. According to (C2) it must try to unify the terms `X` and `fib(2)`, whether the pln function above is present in our program or not. Therefore, if we want to interpret a term in a goal as a pln expression, we have to use a pln goal, that is, a predicate invocation prefixed by a `?` operator: `op(900,fy,?)` (see the Introduction). Then the parameters of the call are *pln expressions*, and they are evaluated.

For example, the goal `? X = fib(2)` first evaluates `fib(2)` to its value, i.e., `2`, provided that `fib//1` is defined as above. (`X` is evaluated to itself.) Then the resulting goal, i.e., `X=2` is called, unifying `X` and `2`.

A *pln expression* can be an *eval expression* (sect. 2.2), a *data term* (sect. 2.3), a *constructor expression* (sect. 3), or a *call to a predefined pln function* (sect. 5).

Syntactically, the pln expressions are always Prolog terms, because PLN is built on the top of Prolog. If a pln expression is a compile-time compound[4], it is an *eval expression* by default, but it may be a *constructor expression*, or a *call to a predefined pln function*, too. The constants and the compile-time variables are *data terms*.

2.2 Eval Expressions

Eval expressions are compound pln expressions. They are function calls invoking arithmetic or user-defined functions (according to our aim: see the Introduction). They are parameterized by pln expressions.

---

basic restriction described in sect. 2.3 is applied. *Innermost* narrowing means that an equation is applied to an expression when the proper subexpresssions of that expression have been evaluated.

[4] The compilation time of a meta-goal is the time when it is called, because it is often transformed before its predicate invocations start to run.

The subexpressions of an *eval expression* are evaluated in the 'leftmost-innermost order' [2]. Therefore an eval expression is tried to be evaluated when its subexpressions have been evaluated successfully.

For example, the eval expression `fib(2)` is evaluated (according to the definition of `fib//1` in sect. 2.1) by the usual *eager evaluation method* as follows:

```
fib(2) -> fib(2-1)+fib(2-2) -> fib(1)+fib(2-2) ->
          1+fib(2-2) -> 1+fib(0) -> 1+1 -> 2
```

2.3 Data Terms

A data term represents itself. No evaluation is done to data terms. Constants (atoms and numbers), variables, and subterms introduced (after compilation[4]) by variable instantiation are always *data terms*.

This later means that an evaluation step "is only performed at a (compound) subterm which is not part of a substitution (introduced by previous unification operations), but belongs to an original (i.e. compile-time[4]) program clause or goal" [2]. Let this restriction be called *basic restriction*, because it is applied in 'basic narrowing' [2]. (This decision follows the usual solution of functional languages, and allows efficient implementation of expression evaluation, without meta-goals depending on the variables of the pln expressions. This is necessary to satisfy (C3).)

In Prolog++ [4], and in earlier versions of PLN [9–12] the atoms (name constants) of the functional expressions were considered symbolic constants denoting other values by default. They were defined by simple equations like `limit=1000.` Programming experience has shown that in most cases the atoms in the pln expressions stand for themselves, and the old semantics is a source of many programming mistakes. Therefore in PLN 5 the atoms are never evaluated, and the *head parts* of the pln clauses must be compound terms (see sect. 2.1). The symbolic constants of a PLN 5 program may be defined as pln facts of a special, user-chosen function. The author's proposal is `v//1`. For example:

```
v(limit)=1000.
v(depth)=20.
```

2.4 A Single Step of Evaluation

Let us suppose, that the principal functor of the current *eval expression*, that is, of the subexpression to be evaluated is f/n ($n > 0$).

If f/n is allowed in an arithmetic expression according to the underlying Prolog system [1, 6], the corresponding step of standard evaluation is performed using the arithmetic of Prolog.
Otherwise a *pln invocation* is executed.

If there exists an appropriate pln function $f//n$, (visible in the source module of the current subexpression), a step of narrowing [2] is performed. First a pln clause whose *head* is unifiable with the current subexpression is selected. Let it

be "*head* `=` *expression* `:-` *condition*." (*condition* may be `true`.) Let the unifying substitution be θ. Next, (if *condition*θ is successful with the substitution φ), the current subexpression is replaced by *expression*$\theta\varphi$. The pln clauses are tried sequentially, and choice points are generated, if there are more candidates. If no pln clause can perform the actual step of evaluation, although the pln function is appropriate in the sense described above, the evaluation step fails and we backtrack to the last choice point, selecting the next alternative. When the last alternative is selected, the choice point is deleted.

Generating choice points is inherited from Prolog. It is used even in the case of `fib//1`: When in the actual implementation [13] (based on SICStus Prolog 3.10 [6]) `fib(1)` is to be evaluated, the first pln clause is tried first, but its *condition*, (actually `1>1`), fails. Then it backtracks, selects the second pln clause, and succeeds in evaluating `fib(1)` to `1`. No choice point remains, because first argument indexing is also inherited from the underlying Prolog implementation [6]. (Therefore the third clause was not considered at the choice point. If `fib(K)` is to be evaluated, and `K\=1`, and `K\=0`, no choice point is generated, because first argument indexing rules out the second and the third pln clauses.)

When the narrowing step above has been performed, *expression*$\theta\varphi$ is usually evaluated further, because *expression* is a pln expression, which fact manifests that functions are most often defined by another functions, and the base cases can be conveniently handled using constructors and quoting (sect. 3, 5.1), if necessary.

The last case of valid evaluation steps is, when a *predicate* is invoked like a pln function. If the current subexpression with the principal functor f/n has an appropriate *predicate* $f/(n+1)$, then this predicate is invoked in the evaluation step: Let us suppose, that the current subexpression is $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ have been evaluated according to the eager evaluation used. Now the predicate invocation $f(t_1, \ldots, t_n, NewVar)$ is performed where $NewVar$ is a fresh variable, and the result of the evaluation is $NewVar$ instantiated by this goal. The step may generate a choice point, or it may even fail and backtrack, as usual.

The last case of valid evaluation steps is allowed to support the use of built-in predicates, and the use of predicates of existing modules in pln expressions. The data flow of these predicate invocations cannot be more general than $(?, ?, \ldots, ?, -)$. (This limitation is resolved with the *block-expressions* in [13].)

If f/n is a non-arithmetic functor ($n > 0$) of a compound [6], and there is neither an appropriate function nor a predicate in the sense decribed above, then the exception `existence_error` is raised.

3 Constructor Expressions

In functional languages many evaluable expressions contain **explicit compound data structures**. However, a compile-time compound of a pln expression is an *eval expression* (sect. 2.2) by default (sect. 2.1). In this section we describe

how to change this default so that the functor of a compund can represent the constructor of an explicit data structure in a pln expression, too.

It is clear that we should not perform evaluation on a pln expression at every position where it is allowed by the basic restriction (sect. 2.3). For example, we must support the pln function concatenating two lists:

```
conc([],List) = List.
conc([Head|Tail],List) = [Head|conc(Tail,List)].
```

There is no evaluation to be performed at the position of the functor `'.'/2`. Consider the pln function for inserting a new item into an unbalanced binary search tree:

```
insert(tree(Root,Left,Right),X) =
    insert(Rel,X,Root,Left,Right) :- compare(Rel,X,Root).
insert(void,X) = tree(X,void,void).

insert(<,X,Root,Left,Right) = tree(Root,insert(Left,X),
                                   Right).
insert(=,X,Root,Left,Right) = tree(Root,Left,Right).
insert(>,X,Root,Left,Right) = tree(Root,Left,insert(Right,
                                   X)).
```

It is clear that no evaluation should be allowed at the positions of the functor `tree/3`. Therefore the functors `'.'/2` and `tree/3` will be *constructors*.

A pln expression is a *constructor expression* iff it is a compile-time compound and its functor is a constructor. A constructor expression is parameterized by pln expressions. When it is called, first its parameters are evaluated. Next, they are replaced by their values, and the resulting term is returned.

The only predefined constructor is `'.'/2` , but `tree/3` can be defined as constructor using a the built-in predicate `constructor/2`. This is a special built-in, the user defines its clauses:

```
constructor(tree,3). % tree/3 is a constructor.
constructor(s,_).    % Every s/N (N>0 integer) is
                       constructor.
constructor(_,0).    % Unnecessary: constants are never
                       evaluated.
```

This means that the functor f/n of a compound is constructor, if the goal `constructor(`f,n`)` is successful with it. Predicate `constructor/2` can be declared multifile and/or dynamic, too.

Constructor definitions are unusual in functional Prolog extensions, and one may ask: "Why is it necessary to define constructors? You could also deduce that each symbol which does not occur outermost in the head of a pln rule/fact is a constructor. (As it is done in [5])" First, we could not. Our precompiler cannot recognize all the pln functions defined in other files and compiled separately. Second, new pln functions would change the meaning of the old pln expressions, (see (C2)), if we adopted this approach.

4 Implementation and Notation

Up till now we have explained semantics. In the actual implementation a pln function is flattened to a Prolog predicate and a pln expression is flattened to a goal sequence according to [2, 5, 9, 13]. For example, `fib//1` (sect. 2.1) is flattened to the following predicate:

```
fib(N,F) :- integer(N), N>1,
        N1 is N-1, fib(N1,F1),
        N2 is N-2, fib(N2,F2), F is F1+F2.
fib(1,1).   fib(0,1).
```

We used the hook predicate `user:term_expansion/2` [6]. Similarly the pln goal `?X=fib(2)` is flattened to the goal `( fib(2,Temp), X=Temp )` using the hook predicate `user:goal_expansion/3` [6]. Consider the predicate

```
p(A) :- ?fib(2*A+1)>2*fib(A)+1.
```

It is flattened into

```
p(A) :- B is 2*A+1, fib(B, C),
        fib(A, D),
        C>2*D+1.
```

These hooks are defined in module `pln` [13], according to (C3).

On the one hand, `fib//1` in sect. 2.1 is much more elegant, and easier to understand than the corresponding predicate above, because "functional notation is more readable for pure functional definitions".[5] On the other hand, this implementation technique implies that pln functions and pln expressions are just notational variants of Prolog predicates and goals. "We believe nonetheless that syntax is important; the power of a good notation is well known from mathematics."[6] After all, this notation often helps the programmer to show the essence of his algorithm.

A flattened call like $f(t_1, \ldots, t_n, NewVar)$ to the pln function $f//n$ is valid, even if written by the user. (It comes from the implementation.) However, $NewVar$ should be a free variable, because a pln function is normally defined with the supposition that its run will not be influenced by some initial value of its returning expression.

Code optimizations, and especially tail-recursion optimization is not performed by our pre-complier. However, if the pln function itself is tail-recursive according to the 'leftmost-innermost' evaluation order, it is flattened into a tail-recursive predicate, too. The reason is that the subgoals of the flattened form of a pln expression come according to the order of evaluation. For example, `f//1` is equivalent to `fib//1` defined in sect. 2.1, and it invokes the tail-recursive `f//3` :

[5] See [8] (Sterling, Shapiro: *The Art of Prolog*, section 3.1 on page 54).
[6] See [8] (Sterling, Shapiro: *The Art of Prolog*, section 8.1 on page 150).

```
f(N) = f(N,1,1) :- integer(N), N>0.
f(0) = 1.

f(K,FN1_K,FN_K) = f(K-1,FN1_K+FN_K,FN1_K) :- K>1.
f(1,FN,_) = FN.
```

(Notice that `?FN1_K=fib(N+1-K)` *and* `?FN_K=fib(N-K)` in `f//3` throughout the recursion.)

The flattened form of `f//1` and `f//3` is the following: (As it can be seen, predicate `f/4` is tail-recursive):

```
f(N,F) :- integer(N), N>0, f(N,1,1,F) .
f(0,1).

f(K,FN1_K,FN_K,FN) :- K>1,
    K_1 is K-1, FN2_K is FN1_K+FN_K,
    f(K_1,FN2_K,FN1_K,FN).
f(1,FN,_,FN).
```

The flattening of constructor expressions is very simple: Their functors are not flattened at all, just their arguments. For example, this is the flattened form of `conc//2` (sect. 3):

```
conc([],List,List).
conc([Head|Tail],List,[Head|TailList]) :-
    conc(Tail,List,TailList).
```

In the flattening process [2, 5, 9, 13], a pln expression is flattened to a goal sequence, and each goal corresponds to a step of evaluation. This method naturally corresponds to our innermost basic narrowing strategy, which is simple and fixed, like the default evaluation order of Prolog goals. According to our current knowledge, lazy evaluation strategy results in much more complex code with considerable run-time overhead, contrasting with (C3), (see [5] for details). It is so, because we are at the level of a simple precompiler, and cannot make sophisticated analysis of the code. This is partly due to the separately compiled parts of the program, and partly due to the scope of the current work. (That is, we flatten every single pln fact/rule independently from each other.)

5 Built-in Functions of PLN

The built-in functions introduce special evaluation techniques. Together with these functions invented by the author, PLN becomes more than an elegant notation. Only four predefined functions of PLN 5 are presented here. See [13] for more details.

5.1 Constructor and quote prefixes

There are some functors that should not be defined as constructors (sect. 3), for example `'+'/2`. If one of these is sometimes used as a constructor, it can be expressed by using the `^` constructor prefix (`op(650,fy,[^,^^])`). The pln expression `^`*exp* is evaluated as the constructor expression *exp*. For example, consider the following symbolic derivation rule:

```
derive(A+B) = ^derive(A)+derive(B).
```

A compound pln expression becomes a data term, if the `^^` prefix is used. This is called *quoting* [4, 7, 9]. The pln expression `^^`*exp* results simply *exp*.

5.2 Meta-Expressions

Meta-programming is possible through *meta-expressions* calculating other evaluable expressions: `apply(`$f, x_1, \ldots, x_n$`)` is the general form of a *meta-expression*.

First it evaluates the pln expressions f and $x_1, \ldots, x_n$. Let their values be g and $y_1, \ldots, y_n$. Now g must be an atom or a compound:

1. If g is an atom, then the function call $g(y_1, \ldots, y_n)$ is evaluated, depending on g/n : If it is
   - a constructor, then the result is the term $g(y_1, \ldots, y_n)$ itself.
   - an arithmetic functor, then the result is the arithmetic value of $g(y_1, \ldots, y_n)$ calculated by the Prolog arithmetic.
   - neither of them, then the value of $g(y_1, \ldots, y_n)$ is calculated by the appropriate *pln invocation* according to sect. 2.4, where $g//n$ may be a user-defined function or $g/(n+1)$ may be any predicate.
2. If g is a compound term, that is, $g = h(z_1, \ldots, z_m)$, then we perform the function call $h(z_1, \ldots, z_m, y_1, \ldots, y_n)$ according to the first case.
3. Otherwise the exception `funcname_error(apply(`$f, x_1, \ldots, x_n$`))` is raised [1, 6, 13].

For example:

```
map_list([X|Xs],Transform) =    % Apply transformation
    [ apply(Transform,X)        % 'Transform' to all the
    | map_list(Xs,Transform)    % items of a proper list.
    ].
map_list([],_Transform) = [].

% Multiply vector V by scalar (number) S.
% V is represented by a proper list of numbers.
mult_by(V,S) = map_list(V,^(*(S))).

% The following operation is a common abstraction of vector
% operations like scalar product and sum of vectors:
```

```
op2v([X|Xs],[Y|Ys],Make,Combine,Null) =
    apply( Combine,
             apply(Make,X,Y),
             op2v(Xs,Ys,Make,Combine,Null)
         ).
op2v([],[],_Make,_Combine,Null) = Null.

% Scalar product of vectors:
mult(V1,V2) = op2v(V1,V2,*,+,0).

% Sum of two vectors:
add(V1,V2) = op2v(V1,V2,+,.,[]).
```

5.3 Conditional Expressions

In order to give a better support to writing deterministic programs, we support special pln expressions: *conditional expressions* of the form

`{` *Condition* `->` *ThenExpression* `;` *ElseExpression* `}`
`{` *Condition* `->` *ThenExpression* `}`

Condition is a Prolog goal sequence, *ThenExpression* and *ElseExpression* are pln expressions. (The *cuts* are local to *Condition*.) If *Condition* is successful, *ThenExpression*, otherwise *ElseExpression* calculates the result of the conditional expression. Only one solution of *Condition* is considered. This is analogous to conditional goals. *ElseExpression* can be a conditional expression without parenthesizing it. *ElseExpression* may also be omitted. If this is the case, and *Condition* fails, the whole expression fails. For example, `fibonacci//1` and `fibon//1` are nearly equivalent. But `fibonacci(N)` raises the appropriate exception if `N` is not a natural number [13], while `fibon(N)` simply fails.

```
fibonacci(N) =
    { \+ integer(N)  -> throw(type_error)
    ; N >= 0 -> fib(N)
    ; throw(constraint_error)
    }.
fib(N) = { N>1 -> fib(N-1)+fib(N-2) ; 1 }.

fibon(N) = { integer(N), N >= 0 -> fib(N) }.
```

6 Conclusions

PLN 5 [13] is a functional extension of ISO Prolog. Its actual implementation is an extension of SICStus Prolog 3.10.

According to the trend of the standardization of Prolog [3], the aim of this project is to generalize Prolog arithmetic to user-defined functions and non-arithmetic expressions. Our constraints are: do not modify the underlying language, support structured programming, and do not lose efficiency.

The project meets its aim and constraints. PLN 5 increases the self-documenting feature of Prolog, because it brings it closer to the traditional notation used by mathematicians.

PLN 5 is designed carefully: Its interconnections with all the other concepts of the underlying language have been examined. The implementation represents a nontrivial engineering task, but its details exceed the scope of this paper.

The current implementation of PLN 5 represents a pragmatic approach, because it is built on the top of a sophisticated and widely used Prolog system, which conforms to the ISO standard [1]. This standard is not yet complete. For example, modules are not yet standardized. Therefore the actual implementation of PLN 5 is integrated with the module system of SICStus. See [13] for the details. Considering the great influence of SICStus on the standardization process, it is believed that the standardization of the remaining features will result in similar solutions to those of SICStus. Summarizing this, we may well say that this project is a step towards the standardization of user-defined functions in Prolog.

References

1. Deransart P., Ed-Dbali A.A., Cervoni L., *Prolog: The Standard (Reference Manual)*, Springer-Verlag, 1996.
2. Hanus M., *The Integration of Functions into Logic Programming*, Journal of Logic Programming, 19&20:583-628, Elsevier Science Publishing Co., Inc., New York 1994.
3. Hodgson J., *WG17 Open Meeting at PACLP 99*, **in:** The ALP Newsleter Volume 12/2, May 1999.
4. Moss C., *Prolog++, The Power of Object-Oriented and Logic Programming*, Addison-Wesley Publishing Company, Wokingham, England, 1994.
5. Naish L., *Adding Equations to NU-Prolog*, Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, Springer LNCS 528, pp. 15-26, 1991.
6. *SICStus Prolog 3.10 User's Manual*, Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden, 2003.
(`http://www.sics.se/isl/sicstuswww/site/documentation.html`)
7. Steele G. L., *Common Lisp: The Language*, 2nd Edition Digital Press, 1990.
8. Sterling L., Shapiro E., *The Art of Prolog* (Second Edition), The MIT Press, London, England, 1994.
9. Ásványi T., *Functions in Full Prolog*, Annales Univ. Sci. Bud. Sec. Comp., 1998.
10. Ásványi T., *Functional Logic Programming with Expression Reduction*, **in:** Pure Mathematics and Applications (PU.M.A.), Budapest University of Economic Sciences, Vol. **9** (1998), No. 1–2, pp. 1–16., Budapest, 1998.
11. Ásványi T., *A Generalization of Arithmetic Expressions in SICStus Prolog*, **in:** 7th International Workshop on Functional and Logic Programming, Proceedings, University of Münster, Germany, June 1998.
12. Ásványi T., *Adding Functions to SICStus Prolog*, **in:** Logic Programming: The 1999 International Conference, The MIT Press, London, England, 1999.
13. Ásványi T., *PLN 5: User's Manual* and *Implementation*
`http://www.inf.elte.hu/~asvanyi/pl/pln/`
Eötvös Loránd University, Budapest, 2003.