Boosting the Quality and Maintainability of Prolog Applications by New Static Analysis Techniques

Thomas Glaß, Martin Müller, Karl Stroetmann, Gerd Venzl

Siemens AG, Corporate Research and Development,
81730 Munich, Germany

Abstract

An approach is presented which supports the development and maintenance of *industrial* Prolog applications that go beyond prototyping. It is particularly well suited for applications that are considered critical with respect to either *reliability* or *maintainability*. The approach is based on an expressive *type system* combined with a *coding standard* for Prolog. Properties of predicates like determinism, unfailing, and termination can be specified by the programmer. The notation is completely transparent to the Prolog interpreter and serves as a formalized *documentation*. Consistency between documentation and code can be checked by automatic static analysis tools integrated in the Prolog Analyzer *PAN*. Our experience shows that programming productivity and program reliability are improved significantly whereas maintenance costs are reduced by an order of magnitude.

1. Introduction and Experiences

In theory, logic programming offers clear advantages over main stream imperative programming. Nevertheless, logic programming as a paradigm and Prolog as a programming language have not been very successful in the software and systems industries. This can be related to the following observation. Prolog can be used as an expressive and extremely flexible high-level language allowing excellent productivity when prototyping and programming algorithms which involve, e.g., complex symbolic manipulations and search strategies. This advantage of Prolog which derives from its expressiveness and flexibility turns into a disadvantage, however, when it comes to quality attributes of software (reliability and efficiency) and software engineering aspects (support for team productivity and maintenance).

Our approach to improve this situation is based on the observation that the majority of predicates in practical Prolog applications do not use the full flexibility and power of the language. We have identified the following issues where a restricted use of the language opens up opportunities for powerful compile-time analysis tools:

Types and Modes Prolog, as an untyped language, does not restrict the arguments in predicate calls. Designer of programs, however, usually have a clear picture of the data for which a program produces reasonable results. The same can be said about dataflow information (moding) which basically declares arguments of predicates to be input or output positions.

Determinism Predicates in Prolog are nondeterministic in general, i.e., they can generate more than one solution. Most predicates in practical programs, however, are in fact deterministic, i.e., they produce at most one solution.

Unfailing If the input data to a predicate are completely unrestricted most predicates will fail for certain inputs. Very often, however, a predicate has been designed not to fail and produce a solution for every well-typed call.

If information on types, modes, determinism, and unfailing is made explicit in program sources, redundant information is added which due to its precise meaning provides a formalized *documentation* and greatly improves *readability*. We have designed a simple *annotation language* which consists of pragmas transparent to the Prolog system.

In addition, our approach offers a *coding standard* which essentially requires a disciplined use of the cut operator. If the coding standard is adhered to, all of the properties mentioned above (i.e. well-typing, correct moding, determinism, unfailing) as well as others (e.g. termination) can be checked automatically by means of easy-to-use and efficient *static analysis* tools. Thereby, complete classes of typical non-trivial coding and design errors in Prolog programs can be located at compile time.

We have implemented such tools and integrated them in the Prolog Analyzer *PAN* which has been used in variety of software projects within Siemens Corporate Research and Development[1]. Some of our experiences with *PAN* are as follows.

Productivity of Prolog programming increases significantly. Very often, non-trivial programs work correctly the first time they are run. The overhead for writing annotations is minor compared to the savings in debugging (*negative overhead*).

Reliability of Prolog programs can be expected to increase accordingly since more errors can be found with *PAN* than with code inspection and testing. However, this can at present not be supported by hard facts since measuring the reliabily of software is a difficult task.

Maintainability of Prolog programs is improved dramatically. Typically, maintenance involves local changes in programs which, however, may affect a large number of predicates not touched explicitely. *PAN* finds out all predicates which need to be changed by pointing to any discrepancy introduced between code and formalized documentation.

Team support is improved by *PAN* since interfaces between different parts of a program can easily be checked for consistency. Typically, common predicates are modified due to design decisions. The correct propagation of changes can efficiently be checked with *PAN*.

Reengineering of existing Prolog code is possible but *not* without cost. If the goal is to modify existing code such that it can be analyzed with *PAN*, experiments have shown that it is not advisable to simply add annotations, only. Instead, parts of the program should be rewritten to make maximal use of the capabilities of *PAN*. If certain predicates of an existing application are exported for use by other code developed with *PAN*, signatures for those predicates have to be provided, only. This requires less work but introduces the risk that discrepancies between specifications and implementations go undetected.

In summary, it is our experience that using *PAN* improves *programming productivity* and *program reliability* quite significantly whereas *maintenance costs* are reduced by an order of magnitude. It is the intention of this paper to give a brief introduction into this technology by focusing on the essentials from a user's point of view. More complete descriptions of *PAN*, its theoretical foundations, and references to related work can be found elsewhere [SGM96, StG95a, StG95b, StG96].

2. Modes and Types

Mode and type information make up the *signature* of predicates. For a predicate *p/n* the signature definition has the form

`%# predicate:` $p(m_1 \tau_1, \ldots, m_n \tau_n)$`.`

where m_i and τ_i are modes and types, respectively. This is an example of the *PAN* annotation language. The pragma '`%#`' makes annotations invisible to the Prolog interpreter and distinguishes them from regular comments.

Modes are a way to classify arguments of predicates as *input* or *output*. In the most simple case we have $m_i \in \{+, -\}$ where '+' denotes input and '–' denotes output. *PAN* supports parti-

[1] Among others, the complete implementation of *PAN* itself has been checked with the tools.

ally instantiated data structures like difference lists and dynamic dictionaries. In this case arguments may be used as input/output and the mode is denoted as '`?`'. Further, *PAN* allows the definition of *multiple modes*. As an example, the predicate `append/3` has the signature

```
%# predicate: append(+list(T),+list(T),-list(T)).
%# predicate: append(-list(T),-list(T),+list(T)).
```

The parameterized type *list(T)* is the set of all lists with elements of type *T* (see below).

Mode correctness can be explained informally as follows. A call of a predicate *p/n* of the form $p(s_1, ..., s_n)$ is defined to be *mode correct* if all input arguments are instantiated. Further, the implementation of *p/n* is defined to be *mode correct* if every call of *p/n* that is mode correct successfully instantiates all output arguments.

As a negative example, consider the following implementation of predicates `p/2` and `q/2`

```
1: %# predicate: p(+atom,-atom).    6: %# predicate: q(+atom,-atom).
2: p(In,Out):-                      7: q(In,Error):-
3:     write(In),                   8:     write(In),
4:     q(Error,Out),                9:     write(Error).
5:     write(Error).
```

which contains two mode errors: The implemention of `p/2` is incorrect since the variable `Error` is not instantiated when `q(Error,Out)` is called. The implemention of `q/2` is incorrect since the output variable `Error` is still uninstantiated when control leaves the body of the clause. *PAN* gives the following messages relating exactly to the errors mentioned:

```
example1.pro:4: dataflow error: no appropriate mode for: user:q/2
example1.pro:4: unbound variables: Error
example1.pro:7: type mismatch: q/2, argno: 2, variable Error
example1.pro:7: expected atom/0, found term/0
```

Note that error messages are preceded by file name and line number associated with the errors.

Types are names for sets of Prolog terms. For example, `int` is the name of the set of all integers and `atom` is the name of the set of all Prolog atoms. The types `int` and `atom` are examples of *basic types*. It is possible to define new, *user defined*, types. An example is the type `colour` consisting of the Prolog atoms '`red`', '`blue`', and '`yellow`'.

```
%# colour:= red + blue + yellow.
```

In general, it is possible to define types using arbitrary function symbols and recursion. As a recursive type definition consider the definition of the type `nat` of natural numbers,

```
%# nat:= 0 + s(nat).
```

A more interesting example is the definition of the type `prop` of *propositional formulae*. Propositional formulae are propositional variables `var(`*N*`)` where *N* is an integer, negations `not(`*P*`)` where *P* is a propositional formula, conjunctions `and(`P_1`,`P_2`)` or disjunctions `dis(`P_1`,`P_2`)`, respectively, where the P_i are propositional formulae. For *PAN* the type `prop` can be defined as

```
%# prop := var(int) + not(prop) + and(prop,prop) + or(prop,prop).
```

This is a recursive *explicit type definition* using function symbols as *type constructors*. Sometimes, *explicit type definitions* are not sufficient because they can have a finite number of constructors, only. Since basic types (`int`, `atom`, ...) have an infinite number of constructors this situation arises if a basic type is extended by new constructors. Therefore, *PAN* offers *type definition by extension*. As an example, if we represent propositional variables by Prolog atoms we would define the type `prop` as

```
%# prop extends atom by not(prop) + and(prop,prop) + or(prop,prop).
```

Well-typedness can be explained informally as follows. Assume an (m+n)-ary predicate *p/(m+n)* with signature

`%# predicate:` $p(+\sigma_1, ..., +\sigma_m, -\tau_1, ..., -\tau_n)$`.`

specifying the first *m* arguments as *input* and the remaining *n* arguments as *output*. The signature also specifies the types of these arguments. The *i*-th input and output arguments are of type σ_i and τ_i, respectively. A call of *p/(m+n)* of the form $p(s_1, ..., s_m, t_1, ..., t_n)$ is *well-typed* if, at the time of the call, the input arguments s_i are instantiated to terms of type σ_i. Further, the program implementing *p/(m+n)* is *well-typed* if for every well-typed and successful call the output arguments t_i are bound to terms of type τ_i.

We proceed with an example showing type errors detected by *PAN*. Then we explain some of the more advanced features of the type concept and illustrate them. A negative example of well-typed programs is the following.

```
1: %# predicate: plus(+nat,+nat,-int).
2: plus(0,X,X).
3: plus(s(X),Y,s(Z)):-
4:     plus(X,Y,Z).
```

In both clauses of `plus/3` the type computed as output is `nat` while, according to the signature, it should be `int`. Note that, a priori, the type `nat` as it has been defined above is not related to the basic type `int`. Therefore we get the error messages

```
plus.pro:2: type mismatch: plus/3, argno: 3, variable X
plus.pro:2: expected int/0, found nat/0
plus.pro:3: type mismatch: plus/3, argno 3:
plus.pro:3: function symbol s/1 not belonging to type int/0
```

Parametric polymorphism introduces type parameters in type definitions and signatures. As an example, consider the definition of the type `list(T)`:

```
%# list(T):= '[]' + '.'(T,list(T)).
```

The type parameter `T` can be instantiated to yield definitions of concrete lists, e.g. integer lists if `T` is substituted by `int`. Since the type `list(T)` is used frequently in Prolog programs *PAN* accepts `T*` as an abbreviation for `list(T)`. An example of a polymorphic signature definition is `append/3` (see above). User-defined parametric polymorphism is less common in practical Prolog applications than one would expect. Besides of the polymorphic list predicates examples of polymorphic types are usually restricted to tree-like structures and graphs.

Inclusion polymorphism introduces subtype relations on types. *PAN* implements an *intensional* or *name based* type system instead of an *extensional* one. In an extensional type system a type σ is a subtype of τ if the set of terms denoted by σ is a subset of the set of terms denoted by τ. For a name based type system a subtype relation between two types σ and τ only exists if

- τ is defined as an *extension* of σ,
- τ is defined as an *extension* of a type ρ and σ already is a *subtype* of ρ, or
- σ is explicitely declared to be a subtype of τ by means of a *subtype declaration* of the form

  `%# subtype:` $\sigma < \tau$`.`

As an example, consider the type `cnf` of propositional formulae in *conjunctive normal form*.

```
%# prop_var:= var(int).
%# literal extends prop_var by not(prop_var).
%# disjctn extends literal by or(literal,disjctn).
%# cnf extends disjctn by and(disjctn,knf).
%# prop:= var(int) + not(prop) + or(prop,prop) + and(prop,prop).
```

Since `cnf` extends `literal` we can use a term of type `literal` anywhere `cnf` is expected. The following program, however, shows that, without an explicit subtype declaration, `cnf` can not be used where `prop` is expected despite of the fact that `prop` contains `cnf` extensionally.

```
1: %# predicate: p(+literal).
2: %# predicate: q(+ cnf).          4: p(X):- q(X).
3: %# predicate: r(+ prop).         5: q(Y):- r(Y).
```

PAN gives the following error message.

```
example2.pro:5: type mismatch: r/1, argno: 1, variable Y
example2.pro:5: expected prop/0, found cnf/0
```

As expected, the predicate `p/2` is well-typed whereas `q/2` is not. The subtype relation which renders the complete program well-typed is

```
%# subtype: cnf < prop.
```

Of course, *PAN* checks that the corresponding sets of terms are indeed in a subset relation. It is a disadvantage of a name based type system that subtype relations which are obvious extensionally have to be declared explicitly. In practical programming, however, a name based type system offers clear advantages which is explained by *type aliasing*.

Type aliasing is a way to give new names to existing types. If we want to define the type σ to be an *alias* for τ this can be achieved via the declaration

```
%# σ == τ.
```

Type aliasing is very useful since it can increase readability of type definitions and predicate signatures as is illustrated by the following example.

```
%# first == atom.  %# town == atom.
%# last == atom.   %# name:= name(first,last).
%# street == atom. %# address:= address(first,last,street,town).
```

However, type aliases have more to offer than increased readability. For the definitions given above we can use terms of types `first` and `last` anywhere we are allowed to use terms of type `atom`. However, we are *not* allowed to use a term of type `first` when a term of type `last` is expected and vice versa. The following predicate `extract/2`.

```
1: extract(address(First,Last,_,_), name(Last,First)).
```

leads to the error message

```
extract.pro:1: type mismatch: extract /2, argno: 2, variable Last
extract.pro:1: expected first/0, found last/0
extract.pro:1: type mismatch: extract /2, argno: 2, variable First
extract.pro:1: expected last/0, found first/0
```

This is a subtle error since we have exchanged the arguments of the functor `name/2`, only.

3. Coding Standard, Determinism, and Unfailing

The determinism and unfailing analysis provided by *PAN* is based on a coding standard and certain rules about how to classify predicates. The general signature of an n-ary predicate *p/n* is

`%#` *Class*`:` *p*($m_1 \tau_1$, ... , $m_n \tau_n$) *Key*`.`

Class serves to classify predicates and is one of the keywords `predicate`, `generator`, `function`, `function_io`, `test`, or `test_io`. For each of these classes specific rules of the coding standard apply. *Key* specifies *unfailing* properties and is empty or one of the keywords `is_complete` or `never_fails`[2]. Unfailing properties will be discussed after we have ex-

[2] Other keywords supported by PAN are `always_fails` and `never_returns` but will not be discussed here.

plained the classes and the corresponding coding standard. To start with, predicates classified as `predicate` are the most general class and not subject to any restrictions at all.

Functional predicates are specified as `function`, `function_io`, `test`, or `test_io`. Functional predicates are *deterministic*, i.e., they must not produce multiple solutions. Therefore, the coding standard for *functional* predicates requires proper usage of the cut operator to avoid backtracking. The general form of a clause implementing a functional predicate is

P:- B_1, !, B_2.

where B_1 and B_2 are conjunctions of literals that contain no cut operator ('`!`'). The rule is that clauses implementing functional predicates have exactly one occurrence of the cut operator with one exception: If the last clause is a fact the cut operator may be skipped.

The different classes of functional predicates are distinguished as follows. Predicates declared as `function` or `test` must not perform any input/output operation whereas there is no such restriction for predicates declared as `function_io` or `test_io`. Predicates declared as `test` or `test_io` have no output arguments, i.e., only test their arguments.

Declarative predicates are the *functional* predicates plus *generators* which are intended to *generate* multiple solutions on backtracking. A typical example of a generator is the predicate `member/2` declared as

```
%# generator: member(-T,+T*) is_complete.
```

On backtracking, this predicate will produce all elements of the list given by its second argument. The coding standard requires that clauses implementing *generators* must not contain any occurrence of the cut operator.

Provided certain further restrictions (see below) the declarative semantics of *declarative* predicates coincides with their operational semantics. The declarative semantics of *generators* is the naive interpretation of Prolog clauses as Horn formulae. The declarative semantics of *functional* predicates is *local* in the sense that clauses are interpreted as formulae

$B_1 \rightarrow (P \leftrightarrow \exists u.B_2)$.

Here u denotes the tuple of all those variables that are *local* to B_2, i.e., those variables that occur in B_2 but do not occur in P or B_1. For this semantics to be correct, the implementation has to be *complete* [StG96]. We will explain the notion of completeness below.

Input/output predicates produce side effects. The same is true for predicates that change global variables or manipulate Prolog databases. The following coding standard is sufficient to guarantee a declarative semantics even in the presence of side effects. The idea is to ensure that side effects do not interfere with the Prolog backtracking mechanism. The rules are:

1. The implementation of *generators* may not contain any input/output predicates.
2. If a *functional* predicate is implemented by a clause of the form

   P:- B_1, !, B_2, L, B_3.

   B_1 must not contain calls to input/output predicates. If a literal L contains calls to input/output predicates B_3 must succeed.

These conditions have to be applied recursively: If a predicate p calls q and q can produce side effects, then p can produce side effects, too. Therefore, the implementation of p has to follow the same rules given above. Such predicates are referred to as *clean* predicates. *PAN* optionally checks for predicates being clean. We clarify this notion with a two example predicates.

```
%# generator: gen_nat(-nat).
gen_nat(0).
gen_nat(s(N)):-
    gen_nat(N),
    write(N).
```

```
%# test: even(+nat) is_complete.
even(0).
even(s(s(N))):-
    !,
    even(N).
```

```
even(s(0)):-
    write('fails'),
    !,
    fail.
```

PAN knows that the system predicate `write/1` produces side effects. Since predicates declared as *generators* are not allowed to have side effects, `gen_nat/1` is not a *generator*. Further, `even/1` is not a *test* because it performs input/output before the cut operator.

Unfailing is specified by means of the attribute `never_fails`. Informally, an (m+n)-ary predicate *p/(m+n)* with the signature definition

`%#` *Class*`:` $p(+\sigma_1, ..., +\sigma_m, -\tau_1, ..., -\tau_n)$ `never_fails.`

is correctly attributed `never_fails` if every call of *p/(m+n)* of the form $p(s_1, ..., s_m, y_1, ..., y_n)$ is successful, provided that s_i is of type σ_i and y_j is a free variable. Of course, this property can not be checked in general. However, given our coding standard, it is possible to state sufficient conditions which imply that a predicate does not fail for calls of the form given above. For *functional* predicates two things have to be checked:

1. For every well-typed call of the predicate there exists a clause matching the input such that the flow of control reaches the cut operator.
2. If the cut operator has been executed none of the succeeding predicates can fail.

Checking the validity of the stated conditions in the general case is still impossible. However, we have implemented a simple theorem prover combining equational reasoning with propositional logic to verify these conditions. Experience has shown that in most cases the conditions can be proven. If it turns out that this is not possible although the corresponding predicate does can not fail, the attribute `assume never_fails` can be used to annotate the predicate.

Completeness is specified by means of the attribute `is_complete`. Essentially, this attribute is a weakening of the attribute `never_fails`. The basic idea is that *intended* failures have to be programmed explicitely by using the predicate `fail/0` (*explicit failure*) such that it can be differentiated from *uninteded* failures resulting from simply 'forgetting' clauses. The attribute `is_complete` is particularly useful for predicates declared as *tests* because for non-trivial tests there will be cases where the test is not successful. If we attribute a test `is_complete` and verify the correctness of this attribution with *PAN* we have established that all case are dealt with. This is clarified by the following example.

```
1: %# test: member_chk(+T,+T*) is_complete.
2: member_chk(X,[Y|_]):-
3:     X == Y,
4:     !.
5: member_chk(X,[Y|Ys]):-
6:     X \== Y, !,
7:     member_chk(X,Ys).
```

```
8: member_chk(_,[]):-
9:     !,
10:    fail.
```

The last clause is necessary for the attribute `is_complete` to be justified. If we drop this clause *PAN* would report the error

```
member_chk.pro:2: incompleteness: member_chk(T1,[])
```

A programming style which makes use of the completeness check leads to programs slightly longer. However, experience has shown that the benefits from the completeness check clearly outweigh this disadvantage. When the attribute `is_complete` has been proven a predicate can not fail any more just because a clause has been forgotten. Furthermore, including information about failure increases readability and maintainability of programs considerably. Finally, if

we want to interpret programs declaratively in the sense of [StG96] *explicit failure* in this sense is a precondition for the declarative semantics to coincide with the operational semantics.

4. More Features of *PAN*

Summarizing, it is our experience that substantial improvements over current Prolog programming practice can be realized by using a methodology like the *PAN* approach. Programming productivity, software reliability and, most notably, software maintenance costs are influenced significantly. Typical for maintenance are modifications of data structures by changing, e.g., arities of existing function symbols or adding new function symbols to a type. Inconsistencies introduced by these changes are effectively found by the type/mode checks and the completeness/consistency checks, respectively.

As a distinguishing feature of our approach we finally emphasize that we do not attempt to introduce a new logic programming language like others do[3] but rather build on the existing, albeit small, industrial user base of Prolog and commercial implementations of the language. Concluding this tutorial introduction to the *PAN* technology we mention some more features which, due to space limitations, can not be described in any more detail.

Termination of Prolog programs can be proven with *PAN* if proper annotations are provided by the user. The annotations are norms on terms and predicates as well as dependency relations between input and output terms of a predicate. Standard norms on terms are provided which can be overwritten by user-defined norms if greater flexibilty is required. User-defined dependency relations are checked by *PAN*, of course. It turned out that the termination analysis is a powerful and easy-to-use feature of *PAN*.

Consistency of the implementation of functional Prolog predicates is related to our notion of *explicit failure*. Whereas, quite informally, *completeness* requires that for every input pattern of a predicate there is a matching clause which reaches its cut operator, *consistency* implies that for every input pattern there is exactly *one* computated result even if the clauses of a predicate are permuted arbitrarily. It turns out that, given a proper programming style, checking for consistency with *PAN* can find very subtle errors not caught by the checks described so far.

References[4]

[SGM96] Karl Stroetmann, Thomas Glaß, and Martin Müller. *PAN* V2.0 - A Tutorial. Siemens AG, 1996.

[SHC95] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. *Logic programming for the real world*. In: Donald A. Smith, Olivier Ridoux, and Peter Van Roy (editors). *Proceedings of the Workshop Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor to Prolog*. pp 83-94. 1995. Available at `ftp://ps-ftp.dfki.uni-sb.de/pub/ILPS-FutureLP/`.

[StG95a] Karl Stroetmann and Thomas Glaß. *A Semantics for Types in Prolog: The Type System of PAN Version 2.0*. Technical Report. Siemens AG, 1995.

[StG95b] Karl Stroetmann and Thomas Glaß. *Augmented Prolog - An Evolutionary Approach*. In: Donald A. Smith, Olivier Ridoux, and Peter Van Roy (editors). *Proceedings of the Workshop Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor to Prolog*. pp 59-70. 1995. Available at `ftp://ps-ftp.dfki.uni-sb.de/pub/ILPS-FutureLP/`.

[StG96] Karl Stroetmann and Thomas Glaß. *A declarative semantics for the Prolog cut operator*. In: Roy Dyckhoff, Heinrich Herre, and Peter Schröder-Heister (editors). *Proceedings of the 5th International Workshop on Extensions of Logic Programming*. Lecture Notes in Computer Science 1050. Springer Verlag. pp. 255-271. 1996.

[3] As an example of such an approach with goals and concepts similar to ours compare the work of the *Mercury* project [SHC95].

[4] Further related work is refered to and discussed in these references.