Pan — A Tool to Analyze Prolog Programs

Karl Stroetmann∗

June 12, 2013

Preliminary Draft

Abstract

This paper describes Pan, a Prolog program analyzer. While Prolog has proved very valuable as a tool for rapid prototyping, there is still some resentment to use Prolog in an industrial context for the implementation of safety critical systems. Part of this resentment is due to the fact that Prolog is an untyped language and therefore lacks the type checking facilities provided by languages like Pascal. Since type checking facilities have proved to be indispensable in order to guarantee a certain quality standard, the use of Prolog as a programming language seems to be excluded for the implementation of safety critical systems.

Fortunately, this situation can be changed. We will propose a programming style that helps to increase the quality of Prolog programs. When using this programming style the user has to decorate his program with mode and type declarations. These declarations can be seen as a systematic way of commenting Prolog programs. Although mere adhering to these conventions would increase the quality of Prolog programs, the potential asset of using this style can be greatly enhanced through the use of a tool specially tailored to scrutinize Prolog programs sticking to these programming conventions. The tool Pan, whose design will be described in this paper, can be used for this purpose.

Besides checking adherence to certain programming conventions, Pan can do a great deal more. Since Prolog is a language that has not only a formal syntax but also a formal semantics, we are able to check additional properties like the completeness and the consistency of programs. We argue that these additional tests, which are not available with conventional programming languages like Pascal and C++, add a new dimension of quality assurance to Prolog that should finally give it a competitive edge over current mainstream programming languages.

To summarize, this paper will propose a programming style for Prolog and will introduce the tool Pan, which is able to check adherence to this style. Usage of Pan will increase the confidence in Prolog programs considerably.

Contents

1 Introduction 2

2 Basic Notions 5

∗Address correspondence to Karl Stroetmann, Siemens AG, ZFE BT SE 11, Otto-Hahn-Ring 6, D–81739 München, Germany. E-Mail: Karl.Stroetmann@zfe.siemens.de.
Experience has shown that Prolog is well suited as an implementation language for the development of software systems which solve problems in the area of theoretical computer science. This is mainly due to four reasons. First, the data type of terms, which is the basic data type of Prolog, has proved to be very flexible. Most data types occurring in applications can easily be modeled by Prolog terms. Furthermore, terms can easily be visualized and are more readable than, e.g. the records of Pascal.

Secondly, matching has shown to be a convenient control mechanism. Determining the control flow via matching is clearly superior than writing cascades of complicated, nested if and case statements, which is necessary in conventional programming languages.

Thirdly, the dynamic memory management provided by Prolog systems relieves the user of worrying about the details of his garbage collections. Memory leaks, a common cause for problems when programming in C++ or Pascal, simply can not occur when programming in Prolog.

Fourthly, since Prolog is interpreted and not compiled it can be used for rapid prototyping. Especially when designing systems performing tasks that are not well understood, Prolog can be used to test new concepts at a very early stage of the software development cycle.

Therefore it comes as no surprise that current state–of–the–art software is developed most cost efficiently in Prolog. Systems like the Sicas–compiler, the CVE environment, the model checker currently developed at Siemens, or the proof compiler SEDUCT, to mention just a few, would be prohibitively expensive without the use of Prolog.
However, since Prolog has a strong theoretical background most people got terribly scared when they first heard of it. Although these fears can in no way be justified since Prolog, although being based on a firm mathematical background, is in practice easier to use than conventional programming languages like C++, people afraid of mathematics searched for excuses not to use Prolog. Their first argument was: “Prolog is not efficient”. While this was certainly true for the first compiler implementations, this argument can no longer be upheld since, on the one hand, current implementations of Prolog compilers are reasonably efficient, and, on the other hand, the performance increases of modern hardware have rendered the efficiency argument invalid. Therefore another argument against Prolog had to be found. This time the community of average\textsuperscript{1} “programmers” — perhaps the noun “hackers” would be more appropriate here — came up with the argument: “Prolog has no type checks.” We do not deny that type checks are a very valuable feature of modern programming languages. But there is no reason why there should be no type checkers for Prolog! Indeed, type checkers for Prolog have already been build and are part of professional Prolog environments, e.g. the Turbo–Prolog\textsuperscript{TM} environment. Since, when using Prolog, the programmer has the opportunity to define his own data types, type checking is even more valuable in Prolog than in conventional programming languages.

However, in order not to endanger the rapid prototyping feature of Prolog, most current implementations of Prolog do not enforce a type check. Fortunately, there is a way of adding type checks to Prolog without sacrificing its rapid prototyping feature. The idea is to construct a separate tool responsible for the type checking. The user has the freedom to use this tool as he desires. He can start by rapid prototyping his problems and use the type checking tool only when things have reached a certain maturity and size and, furthermore, quality becomes an issue. PAN, the Prolog analyzer, is a tool of this kind. When designing PAN we soon found out that, due to the formal nature of Prolog, much more than simple type checking can be done in order to analyze a Prolog program. While tools analyzing programs written in conventional programming languages like C, Pascal, or Fortran can scrutinize their objects only on a rather superficial and syntactical level, much more can be done when analyzing Prolog programs. This is due to the fact that formal techniques can be employed. But there is no reason to get afraid now, its not the programmer who has to use these techniques. Fortunately, due to recent progress in the area of automated theorem proving, it is possible to make the application of formal techniques transparent for the user of PAN.

In the rest of this paper, we will sketch the design of PAN and describe the properties which can be established by PAN. In particular, PAN can analyze the following:

1. Data flow.
2. Completeness.
3. Consistency.

Analyzing the data flow of a Prolog program has two aspects. The first is type checking. The second aspect, which is sometimes known as mode checking, is to guarantee that variables are used in a consistent way, i.e. you may use a variable only once you have assigned a value to it. We will discuss this in more details in Section 4.

The notions of completeness and consistency will be formally defined in Section 5 and Section 6. Here we will only give a rough idea of these notions. The more

\textsuperscript{1}In [1] Edsger W. Dijkstra makes some interesting comments on the concept of the average programmer.
important of these notions is completeness. When writing programs working on complicated data structures, a common problem is the problem of the forgotten case. When writing a program that has, at a certain stage, to consider a large number of different cases, how can we ever be sure that no case has been forgotten? Of course, if one of the common cases has been forgotten, a set of simple test runs should reveal this mistake. But what of the “one–in–a–million–case” that does not show up in the tests? For PROLOG programs sticking to the conventions proposed in this paper it is possible to check whether every case is dealt with. This is due to the systematic way in which terms, the main data structure of PROLOG, are build. Of course, we still have no way of telling whether every case is dealt with properly, but at least we can tell whether none has been forgotten.

The notion of consistency is more subtile than that of completeness. The basic idea is the following. A good deal of the predicates written in a typical PROLOG program really work as functions. Therefore, they should be single–valued. In PROLOG programs this is often trivially achieved by adding cut operators “!”. However, when using a declarative programming style, it is arguable that these cuts should not change the semantics of a program. This is a feature that can be checked with PAN. But let us not get misunderstood: A user of PAN does not have to use a programming style where cuts are used in this restricted way. But when he chooses to do so, then he can later assure himself that he really has stuck to this convention. Chances are good that problems found under this assumption are really due to a flawed analysis of the problem the program under scrutiny has to solve.

We think that the quality level of PROLOG programs, which can be guaranteed through the use of formal techniques as employed by PAN, will give PROLOG a competitive edge over other programming languages, especially when the aspect of safety is a major concern in system development.

Finally, we give a short outlook describing the things to come. The next section briefly introduces some notions used in the rest of this paper. Since we want to check types, it will be necessary for a user of PAN to declare these types. Furthermore, the predicates to be checked have to be declared. Section 3 will discuss this point in detail. Section 4 is devoted to the topic of data flow analysis. Section 5 then presents the methods of showing a program complete. Section 6 deals with the notion of consistency. Finally, Section 7 will discuss the methods found most valuable when proving the verification conditions encountered during the investigation of the completeness and the consistency of PROLOG programs. Later versions of this paper will also include some case studies and a handbook summarizing the commands available in PAN.

\footnote{After all, we are not in the business of magic, or as it is often called nowadays, Artificial Intelligence. Therefore our intention is not to dazzle you with deceptive illusions, but rather to sketch the possible on a sound mathematical footing.}
2 Basic Notions

In this section we will introduce some basic notions which are necessary for the development of the theory in the following sections. Furthermore, we will fix some of our notation and terminology. This section can be skipped by readers who only want to get a rough impression of PAN.

We will use a many–sorted approach to formal specifications. For any set $\mathcal{T}$, a $\mathcal{T}$–sorted set $A$ is a family $(A_S)_{S \in \mathcal{T}}$ of sets indexed by $\mathcal{T}$. The elements of the set $\mathcal{T}$ will be referred to as sorts or as types. If $A$ and $B$ are $\mathcal{T}$–sorted sets, a function $f: A \rightarrow B$ is understood to be a family \( \{f_S\}_{S \in \mathcal{T}} \) of functions indexed by $\mathcal{T}$ such that $f_S: A_S \rightarrow B_S$ for all $s \in \mathcal{T}$.

**Definition 1 (Signature)** A signature $\Sigma$ is a 5–tuple \( \langle T, Cs, Fs, Ps, \text{arity} \rangle \) where:

1. $T$ is a set interpreted as types.
2. $Cs$ is a set interpreted as constant symbols.
3. $Fs$ is a set interpreted as function symbols.
4. $Ps$ is a set interpreted as predicate symbols.
5. The sets $T, Cs, Fs,$ and $Ps$ are pairwise disjoint.
6. $\text{arity}$ is a function assigning an arity declaration to every function and every predicate symbol of the signature. Formally, an arity declaration is a finite string build from the elements of the set $T$. Therefore we have
   \[ \text{arity}: Cs \cup Fs \cup Ps \rightarrow \mathcal{T}^*. \]

If $c \in Cs$ is a constant symbol, then $\text{arity}(c)$ is just the type of $c$.

If $f$ is a function symbol, $\text{arity}(f)$ has the form
   \[ \text{arity}(f) = S_1 \cdots S_n S_{n+1} \text{ with } n > 0. \]

A more convenient notation used in future will be the following
   \[ f: [S_1, \cdots, S_n] \rightarrow S_{n+1}. \]

If $p$ is a predicate symbol such that $\text{arity}(p) = S_1 \cdots S_n$, then we will write this as
   \[ p: [S_1, \cdots, S_n]. \]

For the rest of this section we assume that a signature $\Sigma = \langle T, Cs, Fs, Ps, \text{arity} \rangle$ has been given. Furthermore, in order to be able to define terms, we assume the existence of a $\mathcal{T}$–sorted set $X = \{X_S\}_{S \in \mathcal{T}}$. The members of $X_S$ will be called variables of type $S$.

**Definition 2 (Terms)**

The $\mathcal{T}$–sorted set $\mathcal{T}(\Sigma, X)$ of terms is given via the following inductive definition:

1. All $x \in X_S$ are elements of $\mathcal{T}(\Sigma, X)_S$.
2. If $f$ is a function symbol from $Cs \cup Fs$ such that
   \[ f: [S_1, \cdots, S_n] \rightarrow S_{n+1} \]
   holds, and if for all $i \in \{1, \cdots, n\}$ we have $t_i \in \mathcal{T}(\Sigma, X)_{S_i}$, then $f(t_1, \cdots, t_n) \in \mathcal{T}(\Sigma, X)_{S_{n+1}}$.

The elements of $\mathcal{T}(\Sigma, X)_S$ will be referred to as terms of type $S$. If $t$ is a term containing no variables from $X$, then $t$ is called a closed term. The $\mathcal{T}$–sorted set of closed terms of the signature $\Sigma$ is written as $\mathcal{T}(\Sigma)$. 
Sometimes we need to be able to access *subterms* of terms. Therefore we define the notion of an *occurrence*. Intuitively, an occurrence is just the *address* of a subterm in a term. Formally, an occurrence is a list of positive integers.

**Definition 3 (Occurrence)**
The set $O(t)$ of occurrences of a term $t$ is defined by induction on the structure of terms:

1. $O(x) := \{\[]\}$ for all variables $x \in X$.
2. $O(f(t_1, \ldots, t_n)) := \{\[]\} \cup \bigcup_{i=1}^{n} \{[i|u] : u \in O(t_i)\}$.

The occurrences of $t$ give us the possibility to access the subterms of $t$. Formally, this is done by an inductive definition:

1. $t/[] := t$ for all terms $t$.
2. $f(t_1, \ldots, t_n)/[i|u] := t_i/u$ for all $i \in \{1, \ldots, n\}$ and $u \in O(t_i)$.

We call $t/u$ the subterm of $t$ at position $u$. We can replace subterms of $t$ by other terms. The formal definition is as follows:

1. $t[\[] \leftarrow s := s$.
2. $f(t_1, \ldots, t_n)[[i|u] \leftarrow s] := f(t_1, \ldots, t_i[u \leftarrow s], \ldots, t_n)$ for all $i \in \{1, \ldots, n\}$ and $u \in O(t_i)$.

Intuitively, the term $t[u \leftarrow r]$ results from $t$ by replacing the subterm $t/u$ with the term $r$.

Terms are the basic material to build *well-formed formulae*. First we define the *atomic formulae*.

**Definition 4 (Atomic Formulae)**

1. If $t_1$ and $t_2$ are terms of the same type $S$, then the equation $t_1 = t_2$ is a *first order* atomic formula.

2. If $p$ is a predicate symbol from $Ps$ such that its signature is given by $p: [S_1, \ldots, S_n]$ and if $t_i$ is a term of type $S_i$ for all $i \in \{1, \ldots, n\}$, then $p(t_1, \ldots, t_n)$ is an atomic formula.

A literal is either an atomic formula $P$ or the negation $\neg P$ of an atomic formula $P$. A literal of the form $P$ is called a *positive* literal, a literal $\neg P$ is known as a *negative* literal. Two literals have *opposite signs* if one of them is positive while the other one is negative.

Finally, we can define the set of *well-formed formulae*.

**Definition 5 (Well–Formed Formulae)** The set of well–formed formulae is given by the following inductive definition:

1. true and false are well–formed formulae.
2. Every atomic formula is a well–formed formula.
3. If $F$ and $G$ are well-formed formulae, then $\neg F$, $(F \land G)$, $(F \lor G)$, and $(F \rightarrow G)$ are well-formed formulae.

4. If $F$ is a well-formed formula and $x$ is a variable of type $S$, then $\forall x: S. F$ and $\exists x: S. F$ are well-formed formulae.

In the future we will drop the adjective “well-formed” and just speak of formulae. Unnecessary parentheses will be dropped. This is facilitated by specifying that $\neg$ binds stronger than $\land$ and $\lor$ which in turn bind stronger than $\rightarrow$. To be able to give a meaning to formulae we have to define the notion of a $\Sigma$–structure.

**Definition 6 ($\Sigma$–structure)**

A $\Sigma$–structure $\mathcal{A}$ is a quadruple $\langle (A_S)_{S \in \mathcal{T}}, (c^A)_{c \in Cs}, (f^A)_{f \in Fs}, (p^A)_{p \in Ps} \rangle$, where:

1. $(A_S)_{S \in \mathcal{T}}$ is a $\mathcal{T}$–sorted set. For all $S \in \mathcal{T}$, $A_S$ is interpreted as the set of those elements that are of type $S$.

2. $(c^A)_{c \in Cs}$ is a family of functions such that the following holds: If $c$ is constant of type $S$, then its interpretation $c^A$ is an element of $A_S$.

3. $(f^A)_{f \in Fs}$ is a family of functions such that the following holds: If $f$ is declared as

$$f: [S_1, \ldots, S_n] \rightarrow S_{n+1},$$

then $f^A$ is a function such that

$$f^A: A_{S_1} \times \cdots \times A_{S_n} \rightarrow A_{S_{n+1}}.$$

4. $(p^A)_{p \in Ps}$ is a family of sets such that the following holds: If $p$ is declared as

$$p: [S_1, \ldots, S_n],$$

then $p^A$ is a set such that

$$p^A \subseteq A_{S_1} \times \cdots \times A_{S_n}.$$

Next we have to define the satisfiability relation $|=\mathcal{A}$. To this end we define the interpretation of terms. A term is used to denote an object of our domain of discourse.

Let $\mathcal{A} = \langle (A_S)_{S \in \mathcal{T}}, (c^A)_{c \in Cs}, (f^A)_{f \in Fs}, (p^A)_{p \in Ps} \rangle$ be a $\Sigma$–structure. A valuation $v$ is a $\mathcal{T}$–sorted function $v: (X_S)_{S \in \mathcal{T}} \rightarrow (A_S)_{S \in \mathcal{T}}$. A valuation can be extended to terms. This extension of $v$ will be denoted by $v^\ast$. It is defined via induction:

1. $v^\ast(x) := v(x)$ for all $x \in X$.

2. $v^\ast(c) := c^A$ for all constant symbols $c \in Cs$.

3. $v^\ast(f(t_1, \ldots, t_n)) := f^A(v^\ast(t_1), \ldots, v^\ast(t_n))$ for all function symbols $f \in Fs$ and all terms $t_1, \ldots, t_n$.

If $t$ is a closed term then it is easy to see that the value of $v^\ast(t)$ is, in fact, independent of $v$. Therefore for closed terms we are able to define

$$t^A := v^\ast(t),$$

where $v$ is an arbitrary valuation.

Next we define the notion $\mathcal{A}, v |= P$ for atomic formulae $P$. The notion $\mathcal{A}, v |= P$ is intended to express the fact that the formula $P$ is true in the structure $\mathcal{A}$ under the valuation $v$. For terms $t_1, \ldots, t_n$ and a predicate symbol $p$ we define

$$\mathcal{A}, v |= p(t_1, \ldots, t_n) \quad \text{iff} \quad \langle v^\ast(t_1), \ldots, v^\ast(t_n) \rangle \in p^A.$$

If $t_1$ and $t_2$ are terms of the same type, we define
A substitution is a $\mathcal{T}$-sorted mapping
\[ \sigma : X \rightarrow \mathcal{T}(\Sigma, X). \]
It is lifted to terms homomorphically, i.e.,
\[ \sigma(f(t_1, \ldots, t_n)) := f(\sigma(t_1), \ldots, \sigma(t_n)). \]
A substitution can be applied to literals by treating predicate symbols and the negation symbol $\neg$ as function symbols. A term $t_1$ matches a term $t_2$ via a substitution $\sigma$ iff $\sigma(t_1) = t_2$. Two terms $t_1$ and $t_2$ are unifiable iff there exists a substitution $\sigma$ such that $\sigma(t_1) = \sigma(t_2)$. If $t_1$ and $t_2$ are unifiable then it is well known that there exists a most general unifier. A unifier $\mu$ is more general than a unifier $\sigma$ iff there exists a unifier $\rho$ such that $\rho(x) = \sigma(\mu(x))$ holds for all $x \in X$. The notions of matching and unifiability are easily extended from terms to literals.

If $p$ is a predicate symbol and $t_1, \ldots, t_n$ are terms, then we refer to formulae of the form $p(t_1, \ldots, t_n)$ as predicate calls.

\footnote{The symbol $\not\models$ denotes the negation of $\models$.}
3 How to Specify a Signature in Pan

Pan is a tool designed to guarantee certain properties of a Prolog program. Of course, in order to take advantage of Pan, the user has adhere to a certain programming style, that will be laid down in this section. In particular, Prolog programs have to be decorated by comments describing the modes\footnote{The concept of modes will be defined shortly.} and signatures of the predicates defined. In order to describe these signatures the user has to describe the data structures used in a program. We will first deal with this issue, the issue of modes and signatures will be taken up in the second subsection.

3.1 User Defined Data Types

Although Prolog is an untyped language, the data used by Prolog programs can be classified into different types. Basically, we will distinguish four different classes of data types:

1. System provided data types.
2. Freely generated types.
3. Partial structure types.
4. Quantified partial structures types.

The data types belonging to the first class are those data types which are already hard–wired into the systems while the other three are user defined data types. In the SNI–Prolog system there is just one system data type: the set of all Prolog terms, denoted as term. However, this set can be further divided into disjoint subsets. In SNI–Prolog, the set term is divided into the following subsets:

1. Structures. A structure is anything that can be constructed from a functor and a nonempty list of arguments.
2. Variables.
3. Atoms.
4. Numbers.

These sets are denoted, respectively, as struct, var, atom, and number. In addition to these types there is also the type list, which contains an atom, namely [], and certain structures, e.g. all terms that can be written as [ x | xs ].

The user defined data structures are implemented in terms of data types provided by the system. In conventional Prolog programs the user defined data types are not explicit but nevertheless they are present. The programming style proposed in this paper recommends these data types to be made explicit. For example, consider the following predicate sum_list given below:

\[
\begin{align*}
\text{sum_list}( [ \text{Head} \mid \text{Tail} ], \text{Sum} ) :- \\
\quad \text{!,} \\
\quad \text{sum_list}( \text{Tail}, \text{Tail}_\text{Sum} ), \\
\quad \text{Sum is Head} + \text{Tail}_\text{Sum}. \\
\text{sum_list}( [], 0 ).
\end{align*}
\]
Apparently, this predicate works only when its first argument is instantiated to a list of numbers. We argue that this should be made explicit in a signature declaration of this predicate. For this to work, we first need a notation to describe concepts like “list of numbers”. PAN supports three concepts to define data sets: the freely generated types, the partial structure types, and the quantified partial structure type. We will discuss each of these concepts in turn.

3.1.1 Freely Generated Types

We will start by introducing the concept of a freely generated type. This type is constructed from system types and previously defined types with the help of so called constructors. We will first explain this concept by providing an example and give a formal definition afterwards. Therefore, let us consider again the set of all lists of numbers. It is constructed from the empty list “[]” and the PROLOG cons operator “.”. Hence we can give the following inductive definition of the concept “list of numbers”. If we denote this type as list_of_numbers, then its definition reads:

1. [] ∈ list_of_numbers.
2. If N ∈ number and T ∈ list_of_numbers then ’.’(N, T) ∈ list_of_numbers.\(^5\)
3. Nothing else is in the set list_of_numbers.

In order to give this inductive definition we just had to know the signatures of the constructors “[]” and “.”. In our example “[]” is used as a constant of type list_of_numbers while “.” takes two argument, where the first argument is of type number while the second argument is of type list_of_numbers. We need a convenient notation to express this sort of information. To express the type information used in our example, we write:

```
[] : list_of_numbers.
'.' : [ number, list_of_numbers ] -> list_of_numbers.
```

We need quotes around the dot since otherwise PROLOG would not be able to parse the declaration of the dot operator “.”. After all, we need a notation that can easily be processed by a computer. This is also the reason for sticking to the ASCII character set instead of using mathematical symbols, e.g. we write “→” instead of “→”.

The general syntax for describing the signature of constants is

\[ c : S. \]

Here \( c \) is a meta variable denoting a constant while \( S \) is a meta variable denoting a type. The intended meaning of this declaration is, of course, to express that \( c \) is a constant of type \( S \). The general syntax for describing the signature of a function is

\[ f : [ S_1, \cdots, S_n ] \rightarrow S. \]

This declaration fixes \( f \) to be a function symbol taking \( n \) arguments where the \( i \)-th argument is of type \( S_i \), and producing a result of type \( S \).

When defining freely generated types the range of the constructors used is evident from the context, i.e. the range is always of the type being defined. Therefore, in addition to the notations introduced above, we shall also introduce a notation which merely describes the types of the arguments of a function symbol. For a function \( f \) taking \( n \) arguments of type \( S_1, \cdots, S_n \) we will use the notation:

\[ f( S_1, \cdots, S_n ). \]

Using this notation the following declaration describes the type list_of_numbers

\(^5\)Of course, “’.’(N,T) ∈ list_of_numbers” could also have been written as “[ N | T ] ∈ list_of_numbers” since in PROLOG the latter notation is a shorthand for the former.
together with the signature of its constructors:

\[
\text{list_of_numbers is_freely_generated_by } [ ] \text{ and } '.'( \text{number, list_of_numbers})
\]

The general form of the declaration of a freely generated type \( S \) is as follows:

\[
S \text{ is_freely_generated_by } c_1 \text{ and } \cdots \text{ and } c_m \text{ and } \\
\qquad f_1( S_{11}, \cdots, S_{1k_1} ) \text{ and } \cdots \text{ and } f_n( S_{n1}, \cdots, S_{nk_n} ).
\]

This declaration describes a freely generated type \( S \) which is constructed from the constants \( c_1, \cdots, c_m \) and the functions \( f_1, \cdots, f_n \). The inductive definition for \( S \) is:

1. For all \( i \in \{1, \cdots, m\} \) we have \( c_i \in S \).

2. For all \( i \in \{1, \cdots, n\} \) we have the following:

\[
\text{If for all } j \in \{1, \cdots, k_i\} \text{ it is true that } t_j \in S_{ij}, \text{ then } f( S_{i1}, \cdots, S_{ik_i} ) \in S.
\]

There is one restriction when defining freely generated types: The argument sorts \( S_{ij} \) must be different from the system provided sort \text{var} of all PROLOG variables. We impose this restriction since we will deal with PROLOG terms containing variables separately.

### 3.1.2 Partial Structure Types

Although most user defined data structures used in practice are freely generated types, there are occasions where our restriction regarding the use of PROLOG variables is inconvenient. For example, consider the construction of a dictionary represented by a binary tree. The predicate \text{insert}, which is shown in Figure 1, is used to insert a values under a specified key into a dictionary. The data type used to represent a dictionary is a partial structure since it contains “holes” at its leaves, which can be used to plug in new values. Values inserted under a specified key can not be overridden. In this example, we would use the following declaration to introduce the type \text{dictionary}:

\[
\text{dictionary is_freely_generated_by} \text{ var: dictionary and dict( key, val, dictionary, dictionary )}.
\]
The syntax to introduce partial structure types is the same as that for freely generated types, the only difference being that instead of being generated by constructors alone, a partial structure can also be generated by PROLOG variables used as constructors. However, these variables have to be annotated by a type. Most often this type will be identical to the type defined by that declaration.

### 3.1.3 Quantified Partial Structure Types

The last class of data types the user can define is the class of quantified partial structure types. We will introduce this concept by means of an example. Consider the program shown in Figure 2 implementing the quick-sort algorithm. This pro-

```prolog
% quick_sort(+ list, - list) <--
quick_sort( List, Sorted ) :-
   quick_sort_dl( List, Sorted / [] ).
%
% quick_sort_dl(+ list, - dl) <--
quick_sort_dl( [ X | Xs ], Sorted / Last ) :-
   !,
   split( X, Xs, Small, Big ),
   quick_sort_dl( Small, Sorted_Small / Last_Small ),
   quick_sort_dl( Big, Sorted_Big / Last_Big ),
   Last_Small = Sorted_Big,
   Sorted = Sorted_Small,
   Last = Last_Big.
quick_sort_dl( [], Xs / Xs ).
```

Figure 2: The procedure `quick_sort / 2`

procedure is implemented with the help of difference lists. A difference lists can be viewed as a partial structure together with a pointer which makes the hole in this partial structure easily accessible. The type `dl` of difference lists is given by the following inductive definitions:

1. `Xs / Xs` is a member of `dl`.
2. `[ X | Xs ] / L` is a member of `dl` if `Xs / L` is a member of `dl`.

When contemplating this definition we see that we have something like a partial structure but not quite, because we have wrapped around this structure a function which contains a pointer to the hole in this partial structures. Viewed logically the situation is the same as that with a quantifier in first order logic binding certain variables in a formula. So when defining partial structures with pointers there are really two things to define:

1. First, we need to describe the partial structure.
2. Secondly we need a notation for specifying the binding operator.

The following notation is uses the `λ`-operator to declare difference lists in Pan:

```prolog
lambda_structure dl is list / var(1)
where list is_freely_generated_by
   var(1): list and '.'(number, list).
```
The general syntax for declaring a quantified partial structure $Q$ is given below.

\[
\text{lambda_structure } Q \text{ is } f( S, \text{var}(1), \cdots, \text{var}(l) )
\]

where $S$ is freely generated by

- $c_1$ and $\cdots$ and $c_m$ and
- $\text{var}(1):S_1$ and $\cdots$ and $\text{var}(l):S_l$ and
- $f_1( S_{i1}, \cdots, S_{ik_1} )$ and $\cdots$ and $f_n( S_{i1}, \cdots, S_{nk_n} )$.

This fixes the type $S$ and the type $Q$ to be defined by the following inductive definition:

1. $c_i$ is of type $S$ for all $i \in \{1, \cdots, m\}$.
2. A PROLOG variable $x$ is of type $S$ if there is an $h \in \{1, \cdots, l\}$ such that $S_h = S$.
3. For all $i \in \{1, \cdots, n\}$ the term $f_i( t_1, \cdots, t_{k_i} )$ is of type $S$, provided that for all $j \in \{1, \cdots, k_i\}$ the term $t_j$ is either a term of type $S_{ij}$ or a PROLOG variable. If $t_j$ is a PROLOG variable, there must be an index $h \in \{1, \cdots, l\}$ such that $S_h = S_{ij}$.
4. $f( t, x_1, \cdots, x_l )$ is a term of type $Q$ if the following holds:
   - (a) $t$ is a term of type $S$.
   - (b) $x_1, \cdots, x_l$ are all the PROLOG variable occurring in $t$.
   - (c) Each of these variables does occur in $t$.

\subsection{Mode Declarations of User Defined Predicates}

In the programming style endorsed by PAN predicates used in PROLOG programs have to be classified in one of the following six categories:

1. Tests.
2. Functions.
3. Generators.
4. Input predicates.
5. Output predicates.

We will describe each of these classes in turn.

\subsubsection{Tests}

Conceptually, tests are the simplest predicates. When called they take a set of arguments, do some computation and then either they succeed or they fail, but they will never bind any variables and therefore they compute no explicit result. Furthermore, we postulate that a predicate declared to be a test may not be backtrackable. In order to specify the types of its argument, a predicate used as a test is declared by a statement of the form:

\[
\text{test } p( + S_1, \cdots, + S_n ).
\]

This statement would fix the domain of $p$ to be the set $S_1 \times \cdots \times S_n$. Furthermore, the “$+$” signs specify that $p$ can only be called when all its arguments are bound to terms of the appropriate types. For example, the predicate $\text{member_test}$ whose
definition is given in Figure 3 is declared as follows:

\[ \text{test member_test( term, list ).} \]

Later, when discussing functions, we will introduce the concept of completeness of

\[
\% \quad \text{member_test( term, list ) \leftarrow}
\]

\[
\begin{align*}
\text{member_test( X, [ X | Xs ] ) :-} \\
!.
\text{member_test( X, [ Y | Xs ] ) :-} \\
!,
\text{member_test( X, Xs ).}
\end{align*}
\]

Figure 3: The predicate \text{member_test / 2}

a function. Roughly, this means that a predicate declared to be a function will compute an output for every input, i.e. it will never fail. For tests, the concept of completeness seems, on first sight, not to be very meaningful. After all, a predicate declared to be a test is supposed to fail on certain inputs since otherwise its call is useless! This is certainly true, but we can broaden the concept of completeness to take also tests into account: If a predicate declared to be a test fails on certain inputs, it should do so because of a cut-fail combination and not because there is no clause matching the input under investigation. A predicate \( p \) is declared to be a \text{complete} test by the following command:

\[ \text{test p( S_1, \ldots, S_n ) is_complete.} \]

For example, we can declare the predicate \text{member_test} to be a complete test by writing

\[ \text{test member_test( term, list ) is_complete.} \]

However, the program in Figure 3 is no longer correct with respect to this declaration. We have to change it into the program given in Figure 4 in order to bring it into accordance with this declaration.

\[
\% \quad \text{member_test( term, list ) \leftarrow}
\]

\[
\begin{align*}
\text{member_test( X, [] ) :-} \\
!.
\text{fail.}
\text{member_test( X, [ X | Xs ] ) :-} \\
!.
\text{member_test( X, [ Y | Xs ] ) :-} \\
!,
\text{member_test( X, Xs ).}
\end{align*}
\]

Figure 4: A complete version of the predicate \text{member_test / 2}

\subsection{3.2.2 Functions}

The typical feature of a predicate used as a function is that it uses some of its parameters as input parameters while other parameters are used as output parameters. Although it is actually not required by \text{PAN}, we will assume in the following
that the input parameters precede the output parameters. This convention serves to increase the legibility of declarations. The general form of a function declaration is therefore as follows:

\[
\text{function } p( + S_1, \cdots, + S_k, ? S_{k+1}, \cdots, ? S_l, - S_{l+1}, \cdots, - S_m ) .
\]

For this declaration to be correct, for all \( i \in \{1, \cdots, m\} \) the type \( S_i \) has to be either a user defined or a system defined data type. Additionally, for all \( i \in \{k+1, \cdots, m\} \), the data type \( S_i \) must be a type of the partial structures class. In the declaration above, the arguments at positions \( \{1, \cdots, k\} \) are referred to as input arguments, the arguments at positions \( \{k+1, \cdots, l\} \) are called input/output arguments, and the remaining arguments are called output arguments. The appearance of input arguments is optional but there must be either input/output arguments or output arguments. We call a predicate with no input/output arguments a simple function or sometimes just a function, while the other predicates are referred to as generalized functions. The intention of declaring a predicate as a function is to express the fact that it will always be called with its input arguments instantiated and that, provided the call succeeds, the output arguments will be fully instantiated after that call. This property, which is also referred to as the well–modedness of a program, can be checked automatically by Pan. How this is done will be described in the next section.

A typical function is the append function. It is declared and defined in Figure 5. You may notice that it has been declared as being complete by adding the postfix operator \( \text{is\_complete} \) to its signature definition. Declaring a predicate to be a complete function expresses the expectation that a properly instantiated call of this predicate where all of the output parameters are uninstantiated Prolog variables can not fail. Section 5 discusses how Pan can be used to check such claims. Another important notion of a function is its single–valuedness. We expect a function to return but one value. Therefore functions should not be backtrackable. On a logical level, the single–valuedness of a function is connected with the consistency of certain sets of formula. Methods to check this consistency will be discussed in Section 6.

Logically, a predicate declared to be a simple function still is a predicate, even if its interpretation is single–valued. However, sometimes it is convenient to have a functional notation available. Therefore, if a predicate is declared to be a simple function by a declaration of the form

\[
\text{function } p( + S_1, \cdots, + S_k, - S_{k+1}, \cdots, - S_{k+l}) .
\]

then Pan will declare a function symbol \( p_i \) for every \( i \in \{1, \cdots, l\} \): 

\[
p_i: [ S_1, \cdots, S_k ] \rightarrow S_{k+i}.
\]

We will later see how these functions can be put to use in order to simplify certain proof obligations arising in completeness proofs.
3.2.3 Generators

Generators are functions that are supposed to be backtrackable, i.e. the corresponding functions are no longer single-valued. The syntax to declare a predicate \( p \) to be a generator is the following:

\[
generator \quad p( + S_1, \ldots, + S_k, - S_{k+1}, \ldots, - S_{k+l} ).
\]

Observe that a generator has only input and output positions, input/output positions are not allowed. As for a function, the notion of completeness is useful for generators also. A typical generator together with the definition of the data types involved is given in Figure 6. The predicate call \( \text{count}( n, X ) \) is able to generate

\[
nat \text{ is\_freely\_generated\_by 0 and } s( nat ).
\]

\[
generator \quad \text{count}( + nat, - nat ) \text{ is\_complete}.
\]

\[
\text{count}( s(N), \text{Count} ) := \\
\text{count}( N, \text{Count} ). \\
\text{count}( X, X ).
\]

Figure 6: The generator \( \text{count} / 3 \)

all natural numbers starting from 0 until it arrives at \( n \).

As for predicates defined as functions, it turns out that sometimes a functional notation is quite convenient. Therefore, as in the cases of a function declaration, a generator declaration

\[
generator \quad p( + S_1, \ldots, + S_k, - S_{k+1}, \ldots, - S_{k+l} ).
\]

will, for every \( i \in \{1, \ldots, l\} \), give rise to the declaration of a function symbol \( p_i \) in the following way:

\[
p_i : [ S_1, \ldots, S_k ] \to S_{k+i}.
\]

However, when dealing with these “functions” one always has to observe that the \( p_i \) are not functions in a mathematical sense!

3.2.4 Input Predicates

The declaration of an input predicate \( p \) takes the following form:

\[
input \quad p( + S_1, \ldots, + S_k, - S_{k+1}, \ldots, - S_{k+l} ).
\]

Syntactically, this looks quite like a function declaration. Therefore, why do we bother to introduce this new class of predicates? Well, the difference between an input predicate and a function predicate is that a function will always return the same result when called with the same input arguments. An input predicate, however, will presumably return different results, even when called with the same arguments. To give an example, consider the function \( \text{read} / 2 \) declared as follows:

\[
\text{input} \quad \text{read}( + \text{handle}, - \text{term} ).
\]

At this point you might object that while \( \text{read} / 2 \) certainly is not a function, it could well be regarded as a generator, since generators, too, return different values. However, this argument is not true because, when calling a generator predicate for the first time, it will always return the same result! To validate this, just consider the case of the predicate \( \text{count} / 2 \) defined in Figure 6. Different results are only encountered in backtracking. On the other hand, input predicates are not backtrackable. Rather, they produce their results nondeterministically. Therefore we have to introduce a new class of predicates.
3 HOW TO SPECIFY A SIGNATURE IN PAN

As to completeness, we assume input predicates to be always complete. Therefore there is no special notation with regard to completeness as in the cases of functions and generators. Nevertheless, PAN checks whether the implementation of an input predicate really guarantees its completeness.

3.2.5 Output Predicates

Syntactically, an output predicate is declared as follows:

```latex
output p( + S_1, \ldots, + S_k ).
```

As can be seen from this, the declaration of an output predicate contains only input arguments. This is the same as with predicates used as tests. There is, however, a big difference: An output predicate is not expected to fail. Therefore PAN checks every output predicate for completeness. A typical output predicate is the predicate `write / 2` whose declaration is given below:

```latex
output write( handle, term ).
```

3.2.6 Second Order Predicates

The way second order predicates like the `findall / 3` predicates should be dealt with is still under discussion.

3.2.7 Partially Defined Predicates

Sometimes a predicate is deliberately not defined for all of its possible input arguments, e.g. consider the function `last / 2` given in Figure 7, which computes the “last” element in a list. We can not declare “`last / 2`” as being complete, simply

```latex
function last( + list, - term ).

last( [ X ], X ) :-
    !.
last( [ X | Xs ], Last ) :-
    last( Xs, Last ).
```

Figure 7: The function `last / 2`

because the call `last( [], X )` fails. But since in this case the incompleteness is caused by just one exceptional case, it would still be useful to have a notation able to express this. With PAN we can write the following:

```latex
function last( + list, - term ) is_complete exempting [ last( [], X ) ].
```

The general syntax is the postfix operator `is_complete` followed by the keyword “`exempting`” and a list of all those predicate calls that may fail.

It turns out that this notation covers only one half of the cases of incompleteness occurring in practice. To understand what is missing, consider the case of the predicate `remove / 2` shown in Figure 8. The call `remove( X, L, R )` removes one occurrence of the element `X` from the list `L` to produce the list `R`. However, this works only if `X` is a member of `L`. We argue that if this behaviour is intentional, then the programmer should make this explicit. In PAN this would be done by writing

```latex
function remove( + term, + list, - list ).
```

---

6We disregard here the possibility of failure which is due to limitations of the underlying file system.
function remove(+term,+list,-list).

remove( X, [ X | Xs ], Xs ) :-
  !.
remove( X, [ Y | Ys ], [ Y | Rest ] ) :-
  remove( X, Ys, Rest ).

Figure 8: The function \textit{remove / 2}

\texttt{member( X, L ) implies remove( X, L, _ ).}

The meaning of the "implies" construct is, of course, to express the fact that if the call of the left argument of the \texttt{implies} operator succeeds, then the call of its right argument will succeed also.

### 3.3 Endorsed Programming Style

In this subsection we will give some guidelines concerning the writing of \texttt{Prolog} programs. These guidelines can all be summed up in the following statement:

\textbf{Use a functional programming style where possible!}

We recommend a programming style where every clause defining a non-backtrackable predicate \( p \), i.e. every predicate not declared to be a generator, is given in the following form:

\[ p( t_1, \cdots, t_n ) :\neg T, \!, B. \]  \hfill (*)

If a clause defining \( p \) is given in this form, we require the following:

1. The section \( T \), which will later be referred to as the \textit{test section} of the clause, may contain neither input nor output predicates.

2. Additionally, \( T \) may contain no \texttt{Prolog} variable occurring in a term \( t_i \) such that \( i \) is an output position of the predicate \( p \).

3. \( B \) contains no calls of predicates declared to be either tests or generators.

To ease discussions in the following sections, we will always assume that the terms \( t_i \) appearing in output positions in (*) are variables. However, there is no need for the user of \texttt{PAN} to obey the latter restriction since \texttt{PAN} transforms programs automatically to adhere to this convention. For example, consider the following clause of the \texttt{append} procedure:

\[ \texttt{append( [ X | Xs ], L, [ X | Xs_L ] ) :-}
\]
\[ \texttt{!,}
\]
\[ \texttt{append( Xs, L, Xs_L ).} \]

In \texttt{PAN}, this clause will be represented internally as

\[ \texttt{append( [ X | Xs ], L, Result ) :-}
\]
\[ \texttt{!,}
\]
\[ \texttt{append( Xs, L, Xs_L ),}
\]
\[ \texttt{Result = [ X | Xs_L ].} \]

which is logically equivalent to the original clause.
4 Data Flow Analysis

For Prolog, data flow analysis amounts to more than conventional type checking. To be able to give an algorithm performing data flow analysis we need the notions of typed input and typed output variables of a procedure call. In order to define these notions we in turn need the notion of the set of free, typed variables of a term $t$.

Definition 7 (Free Typed Variables)
The definition of the set $FV(t)$ of the free, typed variables of a term $t$ is given as follows. If

$$t \equiv f(t_1, \cdots, t_n)$$

and the signature of the function symbol $f$ is given as

$$f : S_1 \times \cdots \times S_n \rightarrow S,$$

then $FV(t)$ is defined as:

$$FV(f(t_1, \cdots, t_n)) := FV_{S_1}(t_1) \cup \cdots \cup FV_{S_n}(t_n).$$

Here the function $FV_S(t)$ is defined for any type $S$ and any term $t$ via a case distinction:

$$FV_S(X) := X : S \quad \text{for any Prolog variable } X.$$

$$FV_S(f(t_1, \cdots, t_n)) := FV(f(t_1, \cdots, t_n)).$$

Note that the functions $FV$ and $FV_S$ are defined by mutual recursion. Also note that $FV(X)$ is not defined for a Prolog variable $X$ since it is not possible to determine the type of $X$. The sets $FV(t)$ and $FV_S(t)$ do not contain variables, but rather annotated variables, i.e., pairs of the form $X : S$ where $X$ is a Prolog variable while $S$ is a type symbol.

Definition 8 (Typed Input and Typed Output Variables)
Assume that the signature of the predicate symbol $p$ is given as

$$p( S_1, \cdots, + S_m, - S_{m+1}, \cdots, S_n ).$$

Then for any call $p(t_1, \cdots, t_n)$ of the predicate $p$ the set $FV^+(p(t_1, \cdots, t_n))$ of typed input variables and the set $FV^-(p(t_1, \cdots, t_n))$ of typed output variables can be defined as follows:

$$FV^+(p(t_1, \cdots, t_n)) := \bigcup \{ FV_{S_i}(t_i) : i \in \{1, \cdots, m\} \},$$

$$FV^-(p(t_1, \cdots, t_n)) := \bigcup \{ FV_{S_i}(t_i) : i \in \{m+1, \cdots, n\} \}.$$

Observe that in the definition above we had to use the functions $FV_S$ instead of the function $FV$.

Now we are in a position to formulate an algorithm checking the data flow in the definition of a Prolog predicate. Let us assume that the Prolog program under investigation contains a clause of the following form:

$$p(t_1, \cdots, t_n) : -$$

$$q^{(1)}(t_1^{(1)}, \cdots, t_{m_1}^{(1)}),$$

$$\vdots$$

$$q^{(k)}(t_1^{(k)}, \cdots, t_{m_k}^{(k)}).$$

Then in order to check the data flow of this procedure for $p$ we perform the algorithm described below. This algorithm maintains a variable called $IV$, which contains all of those annotated variables which have already been instantiated. Initially, $IV$ is set to $FV^+(p(t_1, \cdots, t_n))$ since we may assume that all variables declared as input
variables are already instantiated when \( p \) is called. Then, we check for each of the calls of the predicates \( q_i \) in the body of the clause that the input variables of this call are already instantiated, i.e. that they are already present in the set \( IV \). If this is true we know that the instantiation for the call of \( q_i \) is correct and we are therefore entitled to add the output variables of this call to the set \( IV \). Finally, we check whether the output variables of the original call of \( p \) are contained in the set \( IV \). This guarantees that \( p \) instantiates all the variables which it is supposed to compute. The complete algorithm is given in Figure 4.

In addition to the procedure

\[
\text{procedure~check_data_flow}
\]

\[
\text{input: A clause of the form }\]

\[
p(t_1, \ldots, t_n) : \]

\[
g^{(1)}(t^{(1)}_1, \ldots, t^{(1)}_{m^{(1)}}),
\]

\[
\vdots
\]

\[
g^{(k)}(t^{(k)}_1, \ldots, t^{(k)}_{m^{(k)}}).
\]

\[
\text{Signature declarations of all functions and predicates.}
\]

\[
\text{output: yes or no.}
\]

\[
\text{purpose: This procedure checks whether the data flow in the given clause is}
\]

\[
\text{consistent with the declarations of the procedures and functions involved.}
\]

\[
\begin{align*}
\text{begin~procedure} & \quad \text{let } IV := FV^+(p(t_1, \ldots, t_n)); \\
& \quad \text{for } i = 1 \text{ to } k \text{ do} \\
& \quad \quad \text{begin} \\
& \quad \quad \quad \text{if } FV^+(g^{(i)}(t^{(i)}_1, \ldots, t^{(i)}_{m^{(i)}})) \not\subseteq IV \text{ then return( no );} \\
& \quad \quad \text{end;}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{let } IV := IV \cup FV^-(g^{(i)}(t^{(i)}_1, \ldots, t^{(i)}_{m^{(i)}})); \\
& \quad \text{if } FV^-(p(t_1, \ldots, t_n)) \subseteq IV \\
& \quad \quad \text{then return( yes );} \\
& \quad \quad \text{else return( no );}
\end{align*}
\]

\[
\text{end~procedure}
\]

\[
\text{Figure 9: The procedure check_data_flow.}
\]

described above we also have to check that all predicate calls are well typed, i.e. that the arguments of functions and predicates are of the appropriate type. However, this can be done in the same way as it is done in ordinary first order logic and is therefore not described here.

The algorithm given above has to be changed a little bit when dealing with global variables, i.e. when the predicates “get_global / 2” and “put_global / 2” are used. We require every global variable used to be declared and thereby assigned a type. Then, when dealing with global variables, we merely have to check that these variables are of the appropriate type. For this to be possible the body of a clause appearing in a PROLOG program may not contain calls of the form

\[
\text{get_global( X, T ), or}
\]

\[
\text{put_global( X, T )}
\]

where \( X \) is a PROLOG variable, because in these cases it is in general not possible
to determine in advance which global variable is to be accessed.

A further modification of the algorithm has to be made when the procedure under investigation contains input/output variables, i.e. when the declaration of $p$ contains arguments preceded by “?”. The basic idea is to neglect these arguments, since it is not known whether the corresponding terms are instantiated on entry of the procedure call.
Completeness of Predicate Definitions

There are many applications where a user expects a function he has defined to be total, i.e. to return a value on every call. Since in most cases this can be checked automatically, PAN declaration files provide the opportunity to declare a function or relation as complete. This is done by putting the postfix operator is_complete at the end of a declaration. For example, to declare the function append as being complete, the user would use the declaration

\[ \text{function: } \text{append}( + \text{list}, + \text{list}, - \text{list} ) \text{ is_complete.} \]

Declaratively this means that append will always return a value when called with its first and second argument instantiated to lists and the third argument an uninstantiated PROLOG variable. Sometimes a relation is not complete but the incompleteness is caused only by a small number of exceptional cases. For example, if member is defined as

\[ \text{relation: member}( - \text{term}, + \text{list} ). \]

then this relation will return nothing when called as

\[ \text{relation: member}( X, [] ). \]

This case can be dealt with by declaring

\[ \text{relation: member}( - \text{term}, + \text{list} ) \text{ is_complete exempting } \{ \text{member}( ., [] ) \}. \]

The general syntax is here the keyword exempting followed by a list of call patterns for which no output is guaranteed.

There are three common mistakes which cause a predicate definition to be incomplete. These are:

- **premature cuts**,  
- **missing call patterns**,  
- **missing cases**.

We will discuss each of these notions in turn. But before doing so, we will refine some of the notions used so far and introduce some new ones. Furthermore, we will assume each clause defining a given predicate \( p \) to be of the form

\[ p( t_1, \cdots, t_n ) ;: T, !, B. \]  

where \( T \) and \( B \) are conjunctions of literals such that \( T \) does not contain a cut. If \( T \) is empty, then it is simply taken to represent the predicate \text{true}. The same convention applies for \( B \). Additionally, we require every variable to appear at most once in the head of (1) and we assume that only variables appear at output positions of \( p \). These assumptions are made in order to make tests whether two variables are equal explicit. Every PROLOG program can easily be transformed mechanically to satisfy the requirements on variables. Therefore the user is not burdened with writing his programs so that they satisfy these requirements.

Next we define the notion of a call pattern formally. A call pattern for \( p \) is a formula of the form

\[ p( t_1, \cdots, t_n ) \]  

where all terms \( t_i \) appearing at output positions are PROLOG variables. For example, if append is declared as

\[ \text{function: } \text{append}( + \text{list}, + \text{list}, - \text{list} ) \text{ is_complete,} \]

then a program implementing append would presumably use the call patterns

\[ \text{relation: member}( - \text{term}, + \text{list} ). \]

\[ \text{relation: member}( X, [] ). \]

\[ \text{relation: member}( - \text{term}, + \text{list} ) \text{ is_complete exempting } \{ \text{member}( ., [] ) \}. \]

\footnote{This notion is defined below.}
append([], L, R) and append([H | T], L, R).

For a call pattern we require additionally that any variable occurs at most once in it. A program clause
\[ p(s_1, \ldots, s_n) := B, \]
where \( B \) denotes the body of the clause satisfies a call pattern \( p(t_1, \ldots, t_n) \) if and only if the head of the clause matches this call pattern, i.e. there must exist a substitution \( \sigma \) such that
\[ p(s_1, \ldots, s_n)\sigma \equiv p(t_1, \ldots, t_n) \]
holds. In future, if we talk about a program clause matching a call pattern, we will actually mean the instance of the clause corresponding to the particular call pattern under investigation.

Having finished our definitions we are in a position to discuss the reasons for incompleteness of a PROLOG program.

5.1 Premature Cuts

Consider the following implementation of the function \texttt{my_member} which returns \texttt{yes} if the first argument is a member of the second argument and \texttt{no} otherwise:

\[ \text{function: my\_member( + term, + list, - answer ) is\_complete.} \]

\[ \text{my\_member( X, [ Head \mid Tail ], yes ) :-} \]
\[ \text{!,} \]
\[ X = \text{Head}. \]
\[ \text{my\_member( X, [ Head \mid Tail ], Result ) :-} \]
\[ \text{!,} \]
\[ \text{my\_member( X, Tail, Result ).} \]

Obviously, the first cut is misplaced. We call this phenomenon a \textit{premature cut}. More generally, if
\[ p(t_1, \ldots, t_n) := T, !, B \tag{3} \]
\[ \text{is the only clause appearing in the definition of the predicate } p, \text{ then we would like to require that no call of a predicate in } B \text{ can fail, provided of course that the predicate calls in } T \text{ where successful. Formally we require that} \]
\[ \exists \bar{x}_o(T \rightarrow B) \]
holds, where \( \exists \bar{x}_o \) quantifies over all output variables appearing in \( B \). To discuss the most general case, let us assume that the program contains \( m \) clauses satisfying the call pattern \( p(t_1, \ldots, t_n) \). After instantiating these clauses by the substitutions which match the head of the clauses to the pattern under investigation, the relevant section of the program has the following form:\footnote{If there are clauses containing no cut at all, then we can simply add a cut after the last statement of the clause. As long as we are only investigating completeness this cut can have no effect.}
\[ p(t_1, \ldots, t_n) := T_1, !, B_1. \]
\[ \vdots \]
\[ p(t_1, \ldots, t_n) := T_m, !, B_m. \]

Then for all \( i \in \{1, \ldots, m\} \) we demand the following:
\[ \left( \bigwedge_{j=1}^{i-1} \neg T_j \right) \rightarrow \exists \bar{x}_o(T_i \rightarrow B_i) \tag{4} \]
Here $\exists \vec{x}_{\omega}$ quantifies all output variables of the predicates appearing in $B_i$. Let us explain the requirement (4). It is used to check that the cut appearing in the $i$–th clause is not premature. If we investigate this clause, we may already assume that none of the first $i−1$ clauses was successful. This accounts for the conjunction of there test sections in (4). If this conjunction is satisfied, a call of $p$ matched by the pattern under investigation will reach the $i$–th clause. Therefore if the test section $T_i$ of this clause is successful, the program runs past the cut, and we require that none of the predicate calls in the body $B_i$ may fail.

At first sight, due to the appearance of the existential quantifier, verifying (4) seems to be a problem requiring full first order reasoning. However, this is not so if the following requirements are satisfied:

1. Every predicate appearing in the body $B_i$ is declared complete.
2. There are only uninstantiated variables in the output positions of predicate calls appearing in the body $B_i$.

If these conditions are satisfied, then (4) is trivially true. If any of the predicates appearing in the body $B_i$ is complete with the exception of some exempted call patterns, then one simply has show that the negation of the exempted patterns is implied by the conjunction of the $\neg T_j$ and $T_i$. The verification of this implication can usually be done automatically with the help of symbolic evaluation, the use of special axioms describing the data structures under investigation, which will normally amount to the use of the freeness axioms, and propositional reasoning.

### 5.2 Missing Call Patterns

One of the most common reasons for a function not to return any value is a missing call pattern. For example, in the definition of `my_member` given in the last subsection, the pattern

![Pattern](my_member( X, [], R ))

is missing. In this case this will not only result in `my_member` not returning a result for the empty list, but rather `my_member` will never be able to return the result `no`. In order to address problems of this kind, we need some more definitions. Since, in this subsection, we will not be concerned with the internal structure of clauses, we will agree not to distinguish between Prolog predicate symbols and function symbols. However, when viewing a Prolog predicate as a function, only its input positions are relevant, the output positions are neglected. Furthermore, we will choose `bool` as the range of these functions. To give an example, consider the predicate `my_member` defined in the last subsection. When viewed as a function, the signature of `my_member` is given as

`my_member: term × list → bool`.

**Definition 9 (Complete Set of Terms)**

Let $f \in F$s be a function symbol with signature

$f : s_1 \times \cdots \times s_n \rightarrow s$.

A set of terms $T_f$ is called complete for $f$ iff the following conditions are satisfied:

1. All terms $t \in T_f$ can be written as

   $f(t_1, \ldots, t_n)$

   where we have $t_i \in C(\Sigma, X)$ for all $i \in \{1, \ldots, n\}$, i.e., the terms $t_i$ are constructor terms.

2. For every closed term $f(s_1, \cdots, s_n) \in T(\Sigma)$ with $s_i \in C(\Sigma)$ there exists a term $f(t_1, \cdots, t_n) \in T_f$ and a substitution $\sigma$ such that
5 COMPLETENESS OF PREDICATE DEFINITIONS

\[ f(s_1, \cdots, s_n) \equiv f(t_1, \cdots, t_n) \sigma. \]

If \( T_f \subseteq T(\Sigma, X) \) is a set of terms complete for a function symbol \( f \) where \( f \) really is a Prolog predicate, then in order to establish that no call pattern for \( f \) has been forgotten, it is sufficient to show that every term in \( T_f \) can be matched by an existing call pattern. So the only question is to construct a set \( T_f \) which is complete for \( f \). There is one trivial way of doing this: The set

\[ T_f := \{ f(x_1, \cdots, x_n) \} \]

is obviously complete for \( f \). Moreover, it has the advantage of being finite. Unfortunately, however, in most cases this set will be too “general” to be useful. We have to refine it. The next two definitions deals with the concept of refining a set of terms which is complete for a function symbol \( f \).

**Definition 10 (Complete Case Distinction)**

Assume a sort \( s \) to be given. Assume this sort is generated by a set of constructors whose signatures are given as

\[ c_1 : s_{11} \times \cdots \times s_{1m_1} \rightarrow s, \]
\[ \vdots \]
\[ c_k : s_{k1} \times \cdots \times s_{km_k} \rightarrow s. \]

Then the set \( CCD(S) \) is defined as

\[ CCD(S) := \{ c_1(x_{11}, \cdots, x_{1m_1}), \cdots, c_k(x_{k1}, \cdots, x_{km_k}) \} \]

where the \( x_{ij} \) are distinct variables.

**Example:** If the sort \( \text{nat} \) is generated via

\[ \text{zero: } [\] \rightarrow \text{nat} \]
\[ \text{succ: } [\text{nat} ] \rightarrow \text{nat}, \]

then we will have

\[ CCD(\text{nat}) = \{ \text{zero}, \text{succ}(x) \}. \]

**Definition 11 (Refinement of a Complete Set)**

If \( T_f \) is a set of terms complete for a function symbol \( f \), then a refinement \( T_f^+ \) of \( T_f \) can be constructed by the following procedure:

1. Pick a term \( t \) from \( T_f \) which is not closed.
2. Choose an address \( u \in O(t) \) such that \( t/u \) is a variable.
3. If \( s \) is the sort of the variable at \( u \), replace this variable by all terms from \( CCD(s) \):

\[ T_f^+ := T_f \cup \{ t[u \leftarrow p] : p \in CCD(S) \}. \]

Now the question arises as to how far a complete set should be refined. An answer to this question clearly has to take the Prolog program into consideration.

**Definition 12 (Sufficiently Refined)**

A set \( T_f \) of terms complete for a function symbol \( f \) is said to be sufficiently refined with respect to a Prolog program \( \mathbf{P} \) iff for all clauses from \( \mathbf{P} \) of the form

\[ f(s_1, \cdots, s_n) :- B, \]

there is a term \( t \in T_f \) and a substitution \( \sigma \) such that

\[ f(s_1, \cdots, s_n)\sigma \equiv t. \]

(Of course, when writing \( f(s_1, \cdots, s_n) :- B \), we have to neglect output positions.)

It should be clear how to arrive at a set of terms \( T_f \) complete for a function symbol \( f \). We start with the trivial set \( \{ f(x_1, \cdots, x_n) \} \) and refine this set with respect
to the Prolog program under investigation until no more refinement is necessary, i.e., until the set is sufficiently refined. Before we can formulate the algorithm we define the set $T_{\text{def}(f, I^P)}$ as the set of all heads appearing in clauses of $I^P$ defining $f$, i.e.,

$$T_{\text{def}(f, I^P)} := \{ s : s \equiv f(s_1, \cdots, s_n) \land (s : - B) \in I^P \}.$$ 

Now we can describe our algorithm for refinement. This algorithm takes a function symbol $f$ as input. The output is a set $T_f$ which is sufficiently refined with respect to $f$.

1: let $T = \{ f(x_1, \cdots, x_n) \}$;
2: let $S = T_{\text{def}(f, I^P)}$;
3: choose $s \in S$;
4: find $t \in T$ such that $s$ and $t$ are unifiable;
5: % $t$ always exists since $T$ is complete for $f$!
6: choose $u \in O(t)$ such that $t/u \in \mathcal{V}$ but $s/u \notin \mathcal{V}$
7: backtrack to line 3 if there is no such $u$;
8: let $T = T - \{ t \} \cup \{ t[u \leftarrow s] : s \in \text{CCD}(S) \}$;
9: goto 3;

The algorithm stops if it is impossible in line 3 to find a term $s \in S$ with corresponding term $t \in T$ such that in line 6 there is an $u \in O(t)$ with $t/u \in \mathcal{V}$ and $s/u \notin \mathcal{V}$. In this case $T$ is sufficiently refined.

After having constructed a set $T_f$ sufficiently refined for $f$ it is easy to check whether a call pattern for $f$ has been forgotten: We simply have to check whether every term in $T_f$ can be matched by a pattern appearing in the program. If we find a term $t \in T_f$ which is not matched and not exempted either, then we have discovered a problem.

5.3 Missing Cases

Consider a function $\text{condense}$ defined as follows:

```prolog
function: condense( + list, - list ) is_complete.

condense( [], Condensed ) :-
  !,
  Condensed = [].
condense( [ X ], [ X ] ) :-
  !.
condense( [ X, Y | Tail ], Condensed ) :-
  X = Y,
  !,
  condense( [ X | Tail ], Condensed ).
```

When investigating the pattern

```
condense( [ X, Y | Tail ], _ )
```

we realize that the case $X \neq Y$ is missing. In order to analyze this problem we assume that all instances of clauses of the program under investigation satisfying a specified pattern are given as

```
p( t_1, \cdots, t_n ) :- T_1, !, B_1.
```

```
\vdots
```

```
p( t_1, \cdots, t_n ) :- T_m, !, B_m.
```
If we want to prove our program complete, we simply have to demand that one of the tests $T_1, \cdots, T_m$ will be successful. Therefore the following requirement is sufficient:

$$\bigvee_{i=1}^{m} T_i \quad (5)$$

Normally, if (5) holds we will be able to show it by mere propositional reasoning and perhaps some limited amount of symbolic evaluation.

### 5.4 A Sufficient Condition for Completeness

We conclude this section with a theorem.

**Theorem 13** Assume a Prolog program $P$ satisfies the following:

1. The data flow of $P$ is correct.
2. $P$ contains no premature cuts, i.e. (4) is satisfied.
3. The set of call patterns is complete.
4. $P$ has no missing cases, i.e. (5) is satisfied.
5. All predicate calls terminate.

Then the predicates in $P$ are complete, i.e. no predicate call will fail.

With the exception of the last assumption all of the other premisses of this theorem can be checked effectively by Pan. For the sake of incompleteness, a proof of this theorem will not be given.
6 Checking the Consistency

In mathematics functions are single-valued. Therefore we should make sure that a predicate which is declared to be a function is not able to produce alternative solutions on backtracking. An obvious way of ensuring this property is to place a cut as the last statement into every clause used in the definition of that predicate. Checking single-valuedness is then trivial. However, this strategy may obscure some semantical problems which might otherwise result in the many-valuedness of a function. Therefore we will ignore cuts when checking the consistency of a program with PAN.

To check the consistency of a predicate $p$ we first check whether there are two patterns in the definition of $p$ that can be unified by a most general unifier $\mu$. If we do not find two such patterns arising from different clauses, then we already know that the definition of $p$ is consistent. Otherwise, assume that

\[ p( t_1, \ldots, t_m, x_1, \ldots, x_k ) :- B. \quad \text{and} \quad p( s_1, \ldots, s_m, y_1, \ldots, y_k ) :- C. \]

are the two clauses corresponding to these two patterns. Without loss of generality, we have assumed the last $k$ positions to be the output positions. We then require that the results produced by each of these clauses are equal, i.e. we postulate

\[ (7) \]

If true, (7) can always be shown by symbolic evaluation and propositional reasoning, at least in those cases occurring in practice. However, when using this definition to test for consistency, most programs in use will show up to be inconsistent. The reason is best explained with an example. Consider the following two clauses used in the implementation of the function condense introduced in the last section:

\[
\begin{align*}
\text{condense}( [ X, Y | Tail ], R ) :- \\
X = Y, \\
!, \\
\text{condense}( [ Y | Tail ], R ).
\end{align*}
\]

\[
\begin{align*}
\text{condense}( [ X', Y' | Tail' ], R' ) :- \\
!, \\
\text{condense}( [ Y' | Tail' ], R'' ), \\
R' = [ X' | R'' ].
\end{align*}
\]

(We have renamed the variables in the second clause by appending primes in order to prevent name clashes.) In order to establish consistency we would have to verify\(^9\)

\[ X = Y \land \text{condense}( [ Y | Tail ], R ) \rightarrow [ X | R ] = R \]

which is apparently wrong. Nevertheless the implementation of condense is just fine. The reason for our problem is that in the second clause we can assume that $X \neq Y$ holds, since otherwise, due to the cut following the test $X = Y$, we will always stay in the first clause. As a way out of this dilemma we have chosen the following approach: We define the relevant part of the definition of condense as

\[
\begin{align*}
\text{condense}( [ X, Y | Tail ], R ) :- \\
X = Y, \\
!, \\
\text{condense}( [ Y | Tail ], R ).
\end{align*}
\]

\(^9\)For simplicity, we have contracted $\text{condense}( [ Y | Tail ], R ) \land \text{condense}( [ Y | Tail ], R )$ which should appear in the formula below, into a single $\text{condense}( [ Y | Tail ], R )$. 


condense([X', Y' | Tail'], [X' | R']) :-
%Pan not X = Y,
!,
condense([Y' | Tail'], R').

Neglecting for the moment the string “%Pan”, we then have to verify

\[ X = Y \land \text{condense([Y | Tail], R)} \land X \neq Y \rightarrow [X | R] = R \]

which is true since the conjunction of \( X = Y \) and \( X \neq Y \) is contradictory. Had we dismissed the comment in the second clause we would have to pay for it in terms of efficiency. Therefore Pan ignores every comment starting with the symbols “%Pan” when checking consistency. As a further bonus, Pan checks whether it is really safe to assume the tests which have been commented out by proving they follow from the negations of the tests of preceding clauses. Thus Pan provides a way of checking assumptions the user has made regarding the control flow of his program.
7 How to Prove Verification Conditions

Due to lack of time, this section could not be completed.
References