Improving Rule Base Quality with Rule Classification: Prolog Classification Example

By Girish Keshav Palshikar

Introduction
Industrial applications of artificial intelligence, which generally involve large and complex knowledge-bases, are often expensive in terms of time and cost to develop and maintain. These knowledge-bases typically consist of rules, each rule encoding a valuable and reusable piece of domain knowledge or expertise. Although knowledge representation mechanisms other than rules exist – e.g., cognitive maps, Bayesian networks etc. – this article is only concerned with rule-bases. The effectiveness of these real-life systems depends on the nature and contents of these rule-bases.

Knowledge-based systems follow a special development life-cycle that is distinct from typical software development models and historically they have been more expensive to develop and maintain. There is no simple explanation for this nor is there a simple solution to the problem of rule-based system quality. This article explores rule structures and complexities by proposing alternate (but related) rule classifications from different points of view. We propose a thesis, and illustrate through examples, that a rule belonging to a specific and single class is simple and easy to understand while conversely, a rule that cannot be classified cleanly is difficult to understand and maintain. We also present simple programming guidelines for each rule class, that identify some similarity of appearance, interfaces, use, style and that assist each rule's comprehension. We use Prolog as the rule-language, although this general approach is usable by other rule-base engines.

Classification of Rules
It is possible to classify Rules in a Prolog program from various points of view (Table 1). Although many of these classes are already described in the literature and are known to individual Prolog programmers, there are advantages to classifying each rule in a given rule-based system application. We first identify simple programming guidelines for each rule class, ensuring that each rule in a class has some similarity in appearance, interface, use

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Explanation and Classes of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>What are the different styles of the interface for calling a rule? relational, functional, input-only.</td>
</tr>
<tr>
<td>Control</td>
<td>What are the control structures used in the rule body? recursive (pure, backtracking, accumulator-based), imperative</td>
</tr>
<tr>
<td>Logical</td>
<td>Is the rule purely declarative or does it use any non-logical features? declarative, non-deterministic, meta-logical, higher-order, extra-logical, extended logical, DCG.</td>
</tr>
<tr>
<td>Structural</td>
<td>Where does the rule fit in the call graph of the rule-base? top, intermediate, bottom, fact</td>
</tr>
<tr>
<td>Functional</td>
<td>What is the function (task) performed by the rule? condition, search, traversal, generator, random generator, decision, classification, transformation, filter, map, action, query, computational.</td>
</tr>
<tr>
<td>Application-specific</td>
<td>What concept in the application domain does the rule represent? related to concepts in the application domain.</td>
</tr>
</tbody>
</table>

Table 1. Criteria for classification of rules and the associated rule-classes.
Next, understanding the rule structure in a rule-based system facilitates the rule writing (or even re-writing, if necessary) so that each rule (ideally) falls in a single class (when viewed from a specific point of view). This reduces or eliminates the complex and often useless entangling of rules between different classes, thereby simplifying the program's structure and maintenance.

**Classification Using Rule Interface**

Based on a rules input/output interface or the nature of its parameters there are several distinct rule classes. It is possible to classify each parameter in a Prolog rule by its mode as an input parameter, an output parameter or an input-output parameter. An input parameter $X$, denoted as $+X$, must be instantiated when calling the rule. An output parameter $X$, denoted as $\_X$, has the following behavior:

- if $X$ is un-instantiated when calling the rule then it is instantiated by the rule when it finishes successfully;
- if $X$ is instantiated, the rule succeeds only when the computed value of $X$ matches that given.

An input-output parameter $X$, denoted as $\_X$, may or may not be instantiated when calling the rule. However, if it is un-instantiated when calling the rule, it must be instantiated by the rule when it finishes successfully (i.e., when a call to the rule succeeds). A common convention, only using the input/output parameter in those rules that successively return possible values on backtracking, declares the parameter as output if the rule fails on backtracking. A well-defined rule has a unique mode declaration. In a mode declaration for a rule, every parameter to the rule is identified as an input, output or input/output even when there are multiple clauses that define the rule. Some examples of mode declarations are:

```prolog
length(+L,-Len) Len - length of the given list L.
member(X,+L) Successively return elements X from the given list L.
sublist(+SubL,+L) SubL - sublist of the given list L.
```

Most practical rules fall into specific classes based on mode declarations: relational, functional, input-only etc. (see the Interface Criteria in the table). Since rules in a class should have similar mode declarations, if the mode declaration does not fall into a common class, then the rule should be either rewritten to bring it into one specific class or its mode declaration should be justified more rigorously.

**Relational Rules**

A relational rule is a predicate (or Prolog Function) that describes an n-ary or possible multiple relationship between its parameters. For example, Prolog implements the rule that “two persons are siblings if they have a common parent”, as a binary relation between two persons $X$ and $Y$ (given as the rule’s parameters). Note that these rule parameters are input-output. A Prolog rule-base often contains a database of given facts, which are rules without any bodies. For example, consider rules that define relationships such as sibling, uncle, grandfather etc. A fact-base for such a rule-base may contain examples of people and their parents and the rules could be called to derive additional relationships between the people mentioned in the fact-base. The rule `sibling` successively extracts siblings over a fact-base consisting of parents records such as `parents(Meyer, Rita, Robin)` where `Meyer` is the father, `Rita` is the mother, and `Robin` is the daughter.

```prolog
% sibling(?Person, ?Person)
sibling(X, Y) :- parents(F, M, X), parents(F, M1, Y), X 

% sibling(?Person, ?Person)
sibling(X, Y) :- parents(F1, M1, X), parents(F, M2, Y), X \ Y M1 

% sibling(?Person, ?Person)
sibling(X, Y) :- parents(F1, M, X), parents(F2, M, Y), X \ Y, F1 \ F2.
```

Most relational rules allow backtracking so the Prolog predicate `cut` should, in general, be avoided since it stops backtracking. A relational rule typically has separate mutually exclusive sub-rules, each indicating a special condition. A common mistake in writing these relational rules, allowing multiple solutions for the same set of input parameters, enables the invocation of multiple sub-rules upon backtracking. This must be prevented either by the appropriate use of cut or by strengthening the logical conditions in each sub-rule so that they are mutually exclusive.

One problem that occurs in writing relational rules is when un-instantiated parameters are passed when calling: e.g., `sibling(X,Y)`. Although reasonable when the rule allows backtracking, it may not behave correctly if the parameters binding or instantiation is delayed. For example, if the condition $X \ Y$ is the first condition in each sibling rule then the rule fails upon the call `sibling(X,Y)` whereas it sequentially returns the siblings in the form it is written. Hence allow early binding or instantiation of parameters for rules that allow backtracking. Basically, the order of conditions $C_i$ in the body of a rule $A :- C_1, C_2, ..., C_n$ should be critically examined as the results may vary.
depending on the order in which the variables in the rule get instantiated.

**Functional Rules**

A functional rule has one or more input parameters and returns a specific value in an output parameter. For example, a functional rule might return the number of siblings a person has. The following functional rule returns the number of elements in a list.

% length(+List, -NumberOfElements)
length([],0) :- !.
length([H|T], Len) :- length(T,N1),
Len is N1 + 1, !.

This recursive function takes a list and removes the head (H) or left most item in the list and calls itself with the tail (T) as the new input parameter. When the list becomes empty, the top rule succeeds and the functions each complete, one at a time, incrementing Len as they succeed. A functional rule must have at least one input parameter and at least one (preferably exactly one) output (or input-output) parameter. As a matter of style and uniformity, group the input parameters towards the beginning of the parameter list and the output parameters towards the end. Since most functional rules compute the output value only once, a functional rule should use the cut for efficiency and must fail on backtracking.

**Other Types of Rules**

Sometimes rules are neither relational nor functional. For instance, an input-only rule has all input parameters and succeeds if the given values satisfy a specific condition; e.g., the rule

\[
\text{sublist}(+SubL,+L). \\
\text{findall}(\text{member}(\text{NumParents}),-NumParents) \\
\]

computes and returns the number of parents facts in a fact-base.

**Classification using Control Structures**

Ideally, logic programs are devoid of any control for program execution. Prolog supplies a default top-down left-to-right depth-first execution strategy. However, most Prolog programs take advantage of Prolog’s procedural semantics and have a dual declarative as well as procedural nature. To augment the procedural semantics, Prolog offers several control mechanisms: recursion, search and backtracking, 'and fail', looping and all solutions predicates such as findall etc. We can classify Prolog rules based on the control structures that they use.

A recursive rule makes explicit use of recursion. A purely recursive rule does not use any other control structure; the above examples, length and member, are examples of purely recursive rules. An accumulator-based recursion rule makes use of an auxiliary parameter to accumulate intermediate results during recursion; e.g., a rule

\[
\text{sumlist} \\
\]

computes the sum of an integer list is an accumulator-based recursive rule. Rules can also be mutually recursive: such rules are often difficult to understand and maintain. Two rules A and B are mutually recursive if A calls B and B also calls A.

Backtracking is an indirect and auxiliary control mechanism and an important criterion in rule classification. A backtracking rule successively returns solutions, which satisfy the given inputs to the rule. In this sense, backtracking is a control mechanism akin to systematic search and is often used with other

**Prolog References**

Numerous books and articles have been written on Prolog. Here are a few web sites that provide basic Prolog information.

**Prolog Programming a First Course** – an html course intended for those who have some programming experience or have written a few programs in Prolog.
http://cbl.leeds.ac.uk/~paul/prologbook

**Prolog Tutorial** – Intended to be used to help learn the essential, basic concepts of Prolog with sample programs chosen to assist promote the use of Prolog programming in an artificial intelligence course.

**Standard Prolog Source Code** for list processing from references 3 and 4 of this article:

Also see AI and the Net in this issue (p.55) for additional Prolog reference sites.
control mechanisms. For example, member and append are recursive rules which also allow backtracking, referred to as recursive-backtracking rules. Other rules such as the rule sibling above, is a non-recursive backtracking rule.

An imperative rule computes output results and answers using non-logical control looping structures such as repeat, findall etc. or assignment to intermediate variables or counters – e.g. using assert and retract on temporary facts. Although, strictly speaking expression evaluation using the is built-in predicate is an imperative feature of Prolog, we do not consider simple rules containing is as imperative. Specifically, we do not consider recursion as an imperative control structure. Since imperative program often causes side effects; e.g., by using input/output facilities or by changing the fact-bases, their side effects should be clearly identified.

Ideally, each Prolog program should use only one control structure; e.g., a rule should not use both a repeat loop and recursion. Since imperative rules are generally hard to read and test, only use them if absolutely necessary. An imperative rule is often sensitive to the type of input parameters e.g., whether it is an integer, float, atom, string or structure. Such rules should verify that the instantiated inputs have the correct types. An imperative rule using intermediate facts should abolish them before and after the loop to prevent their accidental reuse.

Classification using Logical Contents

We can classify a rule based on the nature and structure of the logical facilities it uses. From this point of view, there are several classes of Prolog rules.

A declarative rule, one written in pure Prolog without using any non-logical features, is purely declarative if the rules meaning (i.e., the answers it generates) does not depend on Prolog’s operational semantics. This includes ordering of clauses in a file, ordering of conditions within a clause, depth-first search strategy, and so on. Otherwise, it is a procedural-declarative rule since its meaning depends on Prolog’s procedural semantics. For example, the meaning of a pure declarative rule remains unchanged even if the order of the logical conditions on the right hand side changes. A pure declarative rule is considered the ideal and one must strive to write as many pure declarative rules as possible – a notoriously difficult task.

A non-deterministic rule has more than one rule head so that for a particular input, it is not always clear which rule head should be used. Only when the computation terminates, do we know the correct choice.

A meta-logical rule is essentially a rule governing other rules and typically uses Prolog’s meta-logical facilities; e.g., var, nonvar, +, -, arg, functor, and clause facilities. Meta-logical rules are powerful and implement different logical systems in Prolog (e.g., fuzzy logic) or complex special-purpose artificial intelligence techniques (e.g., implementation of different search strategies breadth-first or best-first, mean-end analysis etc.). Use meta-logical rules sparingly, if at all, and document them well, along with examples of their uses.

A DCG rule is a production rule in Prolog’s Definite Clause Grammar (DCG) notation The DCG notation allows context-free grammar (production) rules to be written in Prolog. A higher-order rule uses Prolog’s higher-order facilities; e.g., the call primitive and predicates as arguments to rules. An extended logic rule uses additional facilities closer to first order logic; e.g., the not primitive, the ifthenelse primitive etc. An extra-logical rule uses the extra-logical facilities in Prolog; e.g., input/output. Prolog provides a small subset of logic connectives such as AND, OR, IF-THEN. However, most Prolog systems provide an additional set of extended logic facilities, such as if-then-else, for more powerful logical formulation of the rule. Extra-logical facilities allow Prolog programs to perform things outside the logical framework, such as read files and write to the terminal.

Classification using Functionality

Classifying rules, based on the tasks they perform, enables classification on fairly common rule functionality.

A condition rule verifies a condition that the given state or data is expected to satisfy; succeeding if the conditions are satisfied and failing otherwise. For example, rules to check whether a list is empty or a singleton, whether two lists are equal are examples of condition rules. Most condition rules fail on backtracking.

A search rule searches and returns for an item (or a value) that satisfies certain conditions. These searches, typically performed over fact-bases, often permit
backtracking so that successive values are returned. Most often the search strategy is depth-first top-down, which is Prolog’s default search strategy. However, many applications implement more complex search mechanisms such as breadth-first search, best-first search or heuristic search (e.g., A* or AO*).

For example, the member predicate systematically searches a specified list and returns true if the specified item is a member of the list and false if it is not a member.

Traversal rules, a class of rules related to search rules, implements a systematic search, which visits each “state” in a “state-space” exactly once. The traversal is often performed with some purpose, often producing a result at its completion.

A generator rule, which is a kind of search rule, systematically and successively generates (often via backtracking) a value that satisfies specific restrictions. The generated value returns as the rules output parameter; member and append are examples of generator rules. Generator rules allow backtracking and use a search mechanism to generate the values. A random generator rule, which typically uses a random number generator to generate output values, is not considered a search rule.

A classification rule, which analyzes given information (e.g., in fact-bases), succeeds if the information meets specified criteria. For example, one may classify a graph as tree, path, circuit, Hamiltonian path, regular, complete, bipartite etc. For each class, one would write a predicate, which succeeds if the given graph satisfies the required conditions. A graph may belong to more than one class and one classification predicate is not aware of other classes. Therefore, the set of classes is not fixed a priori so that other classification predicates may be added later, without changing those that exist. Most classification rules fail on backtracking. Related classification rules are often collected into groups. Classification rules may appear similar to conditional rules; a distinction that is particularly unclear when the classification is binary (e.g., when a list is empty or non-empty).

A decision (or selection) rule returns a value from a given (or fixed and known) set of values, based on specific criteria. The output parameter typically returns the output value. For instance, a decision

<table>
<thead>
<tr>
<th>Rule with mode</th>
<th>Interface</th>
<th>Control</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>islist(+L)</td>
<td>input-only</td>
<td>¬R, ¬B, ¬I</td>
<td>classification</td>
</tr>
<tr>
<td>member(?X,+L)</td>
<td>relational</td>
<td>R, B, ¬I</td>
<td>search</td>
</tr>
<tr>
<td>append(?L1,?L2,?L3)</td>
<td>relational</td>
<td>R, B, ¬I</td>
<td>generator</td>
</tr>
<tr>
<td>is_prefix(+Prefix,+List)</td>
<td>input-only</td>
<td>R, B, ¬I</td>
<td>condition</td>
</tr>
<tr>
<td>is_suffix(+Suffix,+List)</td>
<td>input-only</td>
<td>R, B, ¬I</td>
<td>condition</td>
</tr>
<tr>
<td>sublist(+L1,+L2)</td>
<td>input-only</td>
<td>R, ¬B, ¬I</td>
<td>condition</td>
</tr>
<tr>
<td>last(+L,-X)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>reverse(+List,-ReversedList)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>has_dup(+L)</td>
<td>input-only</td>
<td>R, ¬B, ¬I</td>
<td>classification</td>
</tr>
<tr>
<td>remdup(+L,-NewL)</td>
<td>functional</td>
<td>RA, ¬B, ¬I</td>
<td>filter</td>
</tr>
<tr>
<td>list_subtract(+L,+ItemL,-NewL)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>list_antisubtract(+L,+ItemL,-NewL)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>emptyL(+L)</td>
<td>input-only</td>
<td>¬R, ¬B, ¬I</td>
<td>classification</td>
</tr>
<tr>
<td>list_length(+List,-Len)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>list_equal(+L1,+L2)</td>
<td>input-only</td>
<td>¬R, ¬B, ¬I</td>
<td>condition</td>
</tr>
<tr>
<td>list_subset(+L1,+L2)</td>
<td>input-only</td>
<td>¬R, ¬B, ¬I</td>
<td>condition</td>
</tr>
<tr>
<td>list_append(+ListOfLists,-L)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>list_adjacent(?X,?Y,+L)</td>
<td>relational</td>
<td>R, ¬B, ¬I</td>
<td>search</td>
</tr>
<tr>
<td>list_get_element(+L,+Index,-Element)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>list_tail(+List,-Tail)</td>
<td>functional</td>
<td>¬R, ¬B, ¬I</td>
<td>filter</td>
</tr>
<tr>
<td>list_front(+List,-Front)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>filter</td>
</tr>
<tr>
<td>delete(+List,+Element,-NewList)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>filter</td>
</tr>
<tr>
<td>ordered(+List)</td>
<td>input-only</td>
<td>R, ¬B, ¬I</td>
<td>classification</td>
</tr>
<tr>
<td>sumlist(+List,?Sum)</td>
<td>functional</td>
<td>RA, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>inner_product(+L1,+L2,-IP)</td>
<td>functional</td>
<td>RA, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>maxlist(+List,-Max)</td>
<td>functional</td>
<td>RA, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>int_range(+M, +N,-List)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>computational</td>
</tr>
<tr>
<td>flatten(+L1,-L2)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>list_union(+L1,+L2,-L3)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>list_intersection(+L1,+L2,-L3)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>transformation</td>
</tr>
<tr>
<td>maplist(+PredName,+L1,-L2)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>map</td>
</tr>
<tr>
<td>applist(+PredName,+L)</td>
<td>functional</td>
<td>R, ¬B, ¬I</td>
<td>map</td>
</tr>
<tr>
<td>R:recursive</td>
<td>RA:recursive-accumulator</td>
<td>B:backtracking</td>
<td>I:imperative</td>
</tr>
</tbody>
</table>

Table 2 shows a classification of the rules in a typical list-processing library, from the interface, control and functional points of view.
rule may select a likely disease from a diseases list. In a program to solve the 8-queen’s puzzle, a rule may select the queens next square to try. In a graph search algorithm, a decision rule may select the next vertex to visit. A selection rule, which is different from a classification rule, selects and returns a chosen value from a fixed set of possible values — fixed and known a priori. A classification rule, which simply succeeds if the given data meets certain criteria, is independent of other class rules.

A transformation rule converts the input to create an output. For example, one transformation rule may remove all parallel edges and self-loops from a given graph and transform it into a simple graph. Another transformation rule might construct another graph — the union of the two given graphs. A third transformation rule might flatten a list if it contains other lists as elements.

Filter rules and map rules are special types of transformation rules. A filter rule generates the output by removing some parts (based on specific criteria) from the inputs. For example, a rule that returns the front of a given list (all elements except the last) is a filter rule. A map rule obtains its output from a one-to-one correspondence to the inputs. For example, a rule whose output list squares the input list (e.g., \([1,2,3,4]\) maps to \([1,4,9,16]\)) is a map rule. Another map rule might replace all pronouns in the given list of words by appropriate nouns.

An action rule is one that causes side effects; e.g., changes in the fact-base using assert and retract. Typically an action rule has the following structure where Condition is a pure Prolog fragment and Action is a Prolog code causing side effects.

\[ R :: \text{Condition}, \text{Action}. \]

In a well-written action rule, the Action part is not expected to fail after the Condition is satisfied. It is a good idea to ensure that action rules follow this structure. Using a cut between Condition and Action often improves efficiency and prevents unexpected back-tracking upon the failure of the Action part.

\[ R :: \text{Condition}, !, \text{Action}. \]

Many times the action rules are not expected to backtrack and hence a cut should be inserted after the Action part.

\[ R :: \text{Condition}, !, \text{Action}, !. \]

Further functional rules classes are defined as follows.

A query rule performs queries on fact bases. For example, a query rule might compute the number of siblings a person has from the parents fact-base. A computation rule computes, which is typically arithmetic in nature, returns a result from the given information and usually fails on backtracking.

Classifications Using Structures

Call-tree analysis of a rule-base can classify rules based on their level in this tree. A fact is a fact in fact-base. A bottom rule does not call any other rule and should therefore be as efficient as possible, since it is potentially used by many rules. A top rule is not called by any other rule. An intermediate rule is neither a top rule nor a bottom rule and its use should be minimized, since these rules are often the most unclear. An interface rule, which interfaces the rule-base with some external systems e.g., database, should be a bottom rule, if possible. A library rule is not specific to anything in the rule-base and is safely reused in other rule-based systems.

Classification Using Application Concepts

It is beneficial to group a program’s rules together based on how they are related to a single concept in the application domain. For example, in a credit evaluation system, the rules could be grouped into the classes of rules related to collateral, project feasibility, company profile, loan history, banking and finance etc. One should clearly identify the application rule classes in a particular application. Each rule in a rule-based system should belong to one application class only, except intermediate or library rules related to common computations.

Conclusions

This article has defined a comprehensive rule classification for a rule-based system from different perspectives. Ideally, each rule should fall in a single class. Otherwise, the rule may be difficult to understand or maintain. We have also presented simple guidelines for writing rules in a specific class.

Adding additional rule classes as well as more classifications enhances our classification framework. We have focused primarily on single-rule classifications. However, this approach can be extended to collecting rules in groups and classifying the groups as well. Finally, our approach can be easily used to build automated tools that assist in the classification process and improve the quality of the rules in the rule-base.

Acknowledgements

I appreciate the support and guidance of Prof. Mathai Joseph. Thanks also to Dr. C. Anantaram and Dr. V. K. Rai. I sincerely thank Dr. Manasee Palshikar for her tireless support and encouragement.

References


Girish Keshav Palshikar, is a scientist with Tata Research Development and Design Centre (TRDDC), India and he can be reached at girishp@pune.tcs.co.in.