Building Libraries in Prolog

Alan Feuer

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

While Prolog has proven useful for writing programs in a variety of domains, it suffers from its lack of support for modularity, particularly for building libraries of routines and data. This paper points out some problems with standard Prolog that make libraries inconvenient. It then describes a solution to those problems based on the concepts of modules and database views.

1. INTRODUCTION

Conceptually, Prolog is a simple, but powerful, language. We have found it useful for representing small databases and for parsing rich languages. However, one place we find Prolog lacking is in its support for modularity; there is no fully satisfactory way to implement libraries of routines or data.

There are two places where we find libraries convenient: to implement common data types and to store domain-dependent data. The common data types are things like lists, sets, and queues. It seems that every program of moderate size reimplements append and member on lists, and half the programs reimplement union and subset on sets.

We have encountered domain-dependent data in the course of building grammars for limited-domains of English. While writing an English front-end to a calendar service, for example, parsers for two or three domains might be implemented: dates, places, people; as well as for the operations on the calendar itself. It is convenient to use the same parts of speech for each of the domains, yet to keep some of the vocabulary distinct. Thus week may be a fine noun in the domain of dates, but not for the domain of places. That is,

```
noun (week)
```

may be true in one domain but not another.

In this paper we will present some of the difficulties in building libraries of Prolog statements. The example of building libraries for common data types will be used because it is easier to illustrate, but the problems and solution are valid for domain-dependent data as well.

2. THE PROBLEMS

This section points out three problems encountered when building libraries of Prolog statements. In what follows, when we speak of standard Prolog, we mean Prolog as defined in Clocksin and Mellish's *Programming in Prolog* (CL081].

2.1 No Local Routines

Occasionally in the process of writing a routine, an auxiliary routine is needed. Consider, as an example, the fast reverse routine from Clocksin and Mellish (p. 141):

```
reverse(L1,L2) :- revzap(L1,[],L2).

revzap([X|L],L2,L3) :- revzap(L,[X|L2],L3).
revzap([],L,L).
```

The sole purpose of revzap is to assist reverse; it should never be called except from reverse. Unfortunately, in standard Prolog all routine names are global, so revzap takes its place in the database alongside reverse. This is especially a problem for library routines as it means that a user of the library must know the names of auxiliary routines so as not to inadvertently reuse them.

A common approach taken in standard Prolog is to use strange names for local routines, sometimes created by including some nonalphabetic character in the name. While this convention works to keep names in libraries separate from those in application programs, it does not prevent name collisions between different library files.

Another approach taken to solve the problem of hiding auxiliary routines is to add the concept of a module to Prolog. Routines are then packaged in a module and those names that should be visible outside the module are exported [EGG82, JON80, PER781 Using this implementation, the definition of reverse might be surrounded by

```
module list
        :— export (reverse).
        definition of reverse and revzap
end list
```

2.2 Consulting Twice Defines Twice

Introducing modules goes a long way towards making libraries usable. Yet, Prolog's weak naming structure is still a problem. Consider an example of building a set using lists. Assume that the module list is available with the routines append and i

```
module set
        :- consult(list).
        :- export(subset).
        :- export(delete).

        subset([H|T],S) :- member(H,S), subset(T,S).
        subset([],S).

        delete(_,[],[]).
        delete(X,[X|T],T).
        delete(X,[H|T],[H|L]) :- delete(X,T,L).
end set
```

The routines defined in the list module, needed for sets, are now in the global name-space.[1] Thus the operations on sets are subset and delete, as well as member which is borrowed implicitly from list. (Unfortunately, append also is borrowed implicitly from list even though we really do not want anyone to *append* to a set.)

Now, suppose queues also are built from lists:

To suggest that ssnset CAN be implemented without using member it to miss the point. Duplicating code if an always available, though not always attractive, alternative to using a common subroutine.

```
module queue
        :- consult(list).
        :- export(add).
        :- export(delete).

        add(X,Q1,Q2) :- append(Q1,X,Q2).

        delete(_,[],[]).
        delete(X,[X|T],Q) :- delete(X,T,Q).
        delete(X,[H|T],[H|Q]) :- delete(X,T,Q).
end queue.
```

While one wouldn't wish to prohibit using both sets and queues in the same program, consulting both set and queue will result in defining the routines of list twice. The result of this for most Prolog implementations is to double the length of each routine, since the new clauses would be appended to the old ones.

2.3 Cannot Overload Routine Names

Notice also in the example that delete is defined for both sets and queues.[2] Since delete is a global name, if both set and queue are consulted in the same program, the clauses for the two deletes would be chained together possibly resulting in one of the implementations being hidden.

Even if it were desirable for the clauses defined in two different files to be chained together, in standard Prolog the clauses cannot be updated selectively; the standard update routine recousult replaces all the clauses in a routine, without regard for where the clauses were defined. This is a handicap particularly in the case where a routine is defined partially by a general rule and partially by context-specific facts.

As an example, consider a general rule about families:

```
sibling(X,Y) :- parent(Z,X), !, parent(Z,Y), neq(X,Y).
```

For a specific family, there may be some people, i.e., the earliest known ancestors, who are known to be siblings but whose parents are unknown. To capture sibling completely, facts of the form

```
sibling(sibl,sib2).
```

are needed. In order to keep things straight, general rules are kept in a file rules and specific facts are kept in the files fam1 and fam2 for two different families. Then

```
:- consult(rules).
:- consult(fam1).
```

sets the context to answer questions about the first family. But there is no easy way to change the context to answer questions about the second family, since

```
:— reconsult(fam2).
```

will retract the general rule about siblings.

3. OUR SOLUTION

We propose a solution to the problems presented in the last section based on two concepts: modules and views.

- The Prolog database is partitioned into *modules;* each module is a collection of routines. A routine is a sequence of clauses, where each clause has the same principal functor in its head.[3] Every routine is contained in some module. A routine may be tagged as private, in which case it is not visible outside the current module.
- Each module sees the database through its *view.* A view is a sequence of modules; each module sees only those routines that are contained in the modules of its view and it sees them in the order in which they occur in the view.

A module can be represented as a list of routines and a routine as a list of clauses. The view for a module is a list of module references. When considering any goal, the view for the module containing the goal is searched top-down, just like the database is searched in standard Prolog. Thus, a view behaves like a dynamic context.

Figure 1 shows the definition and representation of a module list, with the two routines reverse and revzap. reverse contains one clause and revzap contains two. The view for list is list followed by builtin. In the program, rather than explicitly declaring routines public, we use the operator "$" to hide a routine name. Thus, "Srevzap" is a routine that is private to list.

When a module is created it is given a view of the database that includes itself plus the builtin routines (in the module builtin). In our Prolog, builtins are those routines coded inside the interpreter itself. Another module, called standard, contains many of the routines commonly thought of as builtin but actually coded in Prolog, such as *atomic* and *not.*

The directive

```
use(X)
```

adds the module associated with the file *X* to the view of the current module. There is a one-to-one correspondence between modules and files. A file contains Prolog source text and a module contains Prolog objects; modules exist only within the interpreter. The file-name and the module-name are the same. If *X* is not yet in the database, the file *X* is read from the file system and the module *X* is built. Here is how set might be implemented given modules, view, and use:

```
:- use(list).

member(X,S) :- list$member(X,S).

subset([H|T],S) :- member(H,S), subset(T,S).
subset([],S).

delete(_,[],[]).
delete(X,[X|T],T).
delete(X,[H|T],[H|L]) :- delete(X,T,L).
```

The modules in the view for set are set, builtin, and list, in that order. Notice that where the previous implementation of set implicitly relied upon member for list, this implementation does so explicitly. The "$" in "list$member" constructs an absolute reference to the routine member in the module list.[4] Queue is defined similarly to set, beginning with a use directive.

Next, consider a program, p, that uses both sets and queues. It might begin with two use directives:

```
:- use(set).
:- use(queue).
```

In p, member would refer to the routine member defined in the module set. delete, unadorned, likewise refers to delete in set. queueSdelete refers to delete in queue, append would be an error, as would be listSappend, since these routines are defined in list, which is not visible to p.

Finally, the case of selective reconsulting is easily handled. Using the modules rules, faml, and fam2 with sibling being defined in each, the context for the first family is established similarly to before:

```
:- use(rules).
:- use(fam1).
```

2. The same definition of delete could be used for both sets and queues by not taking advantage of the knowledge that element! only occur once in a aet. But, again, this it besides the point.

3. Routine* are not demarcated syntactically in Prolog. Since, by definition, a routine is the sequence of clauses with a given principal functor in their head, it is impossible to have two routines with the tame name (i.e., the same principal functor) in the same module.

4. The expression ''SX" used to reference an *X* that is private can be viewed as an absolute reference to A' in the current module.

Definition

```
reverse(L1,L2) :- $revzap(L1,[],L2).

$revzap([H|T],L2,L3) :- $revzap(T,[H|L2],L3).
$revzap([],L,L).
```

Representation

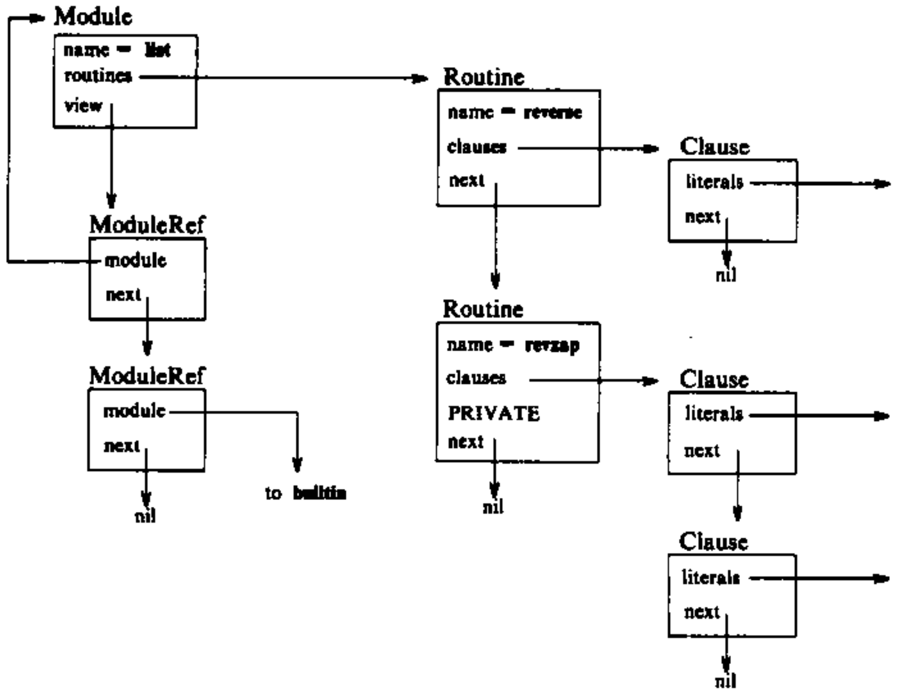

Figure 1. Definition and Representation of the Module list

To change the context to reflect the second family, we introduce the directive remove which removes all of the clauses associated with a given module:

```
:— remove (fami).
:-  use(fam2).
```

Since clauses are associated with a module, the general rule for sibling that resides in rules is untouched.

4. SUMMARY

Prolog is increasing in popularity because it embeds the elegance and power of logic in an easy to use language. Unfortunately, libraries of Prolog routines are not easy to build because the naming structure of Prolog is too simple. In particular, we have identified three problems: absence of local routines, consulting a file more than once, and overloading routine names.

Our solution to these problems rests on the ideas of modules and views. A local routine is a routine visible only within the module in which it is defined. Instead of *consulting* files, we *use* modules; use may consult a file if it isn't already in the database. Finally, a routine is accessed by its name and by its module, thus there is no conflict when two routines have the same name in different modules—since a view is an ordered list of modules, the routine occurring first in the view is the one referenced. The other may be accessed by using an absolute reference.

ACKNOWLEDGEMENT

I am grateful to Shamim Naqvi and Claude Sammut for their comments on a draft of this paper.

REFERENCES

CL081	W.F. Clocksin and C.S. Mellish, *Programming in Prolog,* Springer-Verlag, 1981.
EGG82	P.R. Eggert and D.V. Schorre, "Logic enhancement: a method for extending logic programming languages", Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.
JON 80	S.B. Jones, "Structured programming techniques in Prolog*', Proceedings fo the Logic Programming Workshop, July 1980.
PER78	L.M. Pereira, F.C.N Pereira, and D.H.D. Warren, *User's Guide to DECsystem-10 Prolog,* Dept of Artificial Intelligence, University of Edinburgh, September 1978.