Logic Programming and Deductive Databases

Chapter 9: Modules in Prolog (Short Introduction)

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2023

http://www.informatik.uni-halle.de/~brass/lp23/

Objectives

After completing this chapter, you should be able to:

- use modules in Prolog.

Contents

Reasons for Using Modules (1)

- Normally, all predicates in a Prolog program are in a single namespace.
- For small programs, this is no problem. For large programs, there can be name clashes.

  At least, most Prolog systems will print an error message, because the clauses of that predicate will not be consecutive in the program.

- Every predicate can be called from every other predicate in the program.
- However, auxiliary predicates are quite common in logic programs. It should be easy to change these, because they are used only in a small part of the program.

  SWI-Prolog has a cross-referencing tool: [https://www.swi-prolog.org/gxref.html]

Reasons for Using Modules (2)

- It is a general principle in software development to clearly separate interface from implementation, so that one can
  - change/improve the implementation, and
  - keep the interface stable.

    The auxiliary predicates are only part of the implementation of a module. One does not need to understand them when using the interface predicates (predicates exported from the module). At least, when the interface predicates are well documented.

- Thinking about modules while developing a program can help to achive a better structure.

  When one develops a large program, one always has some structure of the program into components in one's mind. With modules, one can make this explicit, and document it for other programmers who later have to understand the program. Modules are another layer of abstraction above the predicates.

Portability Problems

- The first Prolog systems did not contain modules, and Prolog systems differ in their support of modules.
- Since 2000, there is a ISO standard for modules in Prolog (ISO/IEC 13211-2), but SWI Prolog and many other Prolog systems do not follow this standard.
- SWI Prolog as well as SICStus, Ciao and YAP were inspired by the module support of Quintus Prolog.
- Günter Kniesel (University of Bonn) writes: "Fear of non-portability should not stop you from using modules: An unmanageable, badly structured program is unusable even on a single platform."

[https://sewiki.iai.uni-bonn.de/_media/teaching/lectures/alp/2012/08_-_modules_and_objects.pdf]

Basics of Modules in Prolog

- In Prolog, modules are namespaces for predicates.
- I.e. predicate `p/1` in module `mod1` is different from predicate `p/1` in module `mod2`.
- However, atoms and functors (data terms) are not distinguished between modules. E.g. `a` in code from module `mod1` unifies with `a` in code from module `mod2`.

Contents

Defining a Module

- Even without modules, one can structure a large Prolog program in several source files, and consult all these files in a "main file".

  The user of the program then only needs to consult the "main file", this consults all the other source files (similar to `#include` in C).

- A module is simply a source file that starts with a "module declaration":

```
:- module(Name, [Pred1/Arity1, ...]).
```

  - "`Name`" is the name of the module.
  - "`Pred1/Arity1, ...`" are the exported predicates.

- The source file name should be "`Name.pl`", i.e. the module name with the standard extension for Prolog.

Using a Module

- To make predicates from another module available in the current module, one writes:

```
:- use_module(Name, [Pred1/Arity1, ...]).
```

- Here "`Name`" is the name of the source file (without extension), not the module name, but normally, these are identical.

  The file name is needed, because `use_module` loads the program code of the module similar to `consult` (if the module is not already loaded). In principle, it is not required that the source code of module "`a`" is in file "`a.pl`". But if one follows this convention, one does not have to remember that `use_module` really needs the file name.

- In order to make all exported predicates from module "`Name`" available, one can simply write:

```
:- use_module(Name).
```

Name Conflicts in Imports

- One cannot import two predicates with the same name and arity from two different modules.
- Therefore, it is possible to rename a predicate when it is imported:

```
:- use_module(moduleA, [p/1 as pa]).
```

- Since the list of exported predicates of a module might be extended later, it is safer to explicitly list the imported predicates in the `use_module` declaration.

  Otherwise, one might later suddenly get an error message for a name conflict in the imports, and has no documentation, from which of the two modules the predicate originally was used.

- It is possible to redefine an imported predicate within the module, but this gives a warning (might be switched off).

Special Modules

- All built-in predicates are defined in the module "`system`".
- All predicates defined outside modules are implicitly in the module "`user`".

  The two module names are reserved. There cannot be two modules with the same name (i.e. there is a flat namespace for modules). One can query the current module with `context_module(M)`.

- SWI Prolog has an "`autoload`" feature to automatically load undefined predicates from certain modules.

  [https://www.swi-prolog.org/pldoc/man?section=module-autoload]

- The "`user`" module autoloads from "`system`", all other modules autoload from "`user`" and "`system`".

  This has the effect that predicates defined outside modules (i.e. in "`user`") are global, i.e. can be used in every module.

Explicit Module Names

- Internally, predicates are identified by module name, predicate name, and arity.
- One can explicitly refer to a predicate in a module with the notation

  `module:pred(...)`.

- This is even possible for predicates that were not exported!

  I personally consider this as a bad design decision. However, it has some advantages, e.g. when testing and debugging a program. One can call from the top level prompt any predicate in the program and is not limited by module boundaries.

- Of course, it is normally bad style to call a predicate with explicit module prefix.

  Effectively, the export list of a module declaration only documents what the author of the module intended to export. The user can override this.

Meta Predicates (1)

- When a data term becomes a predicate call, a module must be added to the function symbol in order to use it as a predicate. Normally the current module is used.
- If one wants to define and export a meta predicate like `findall`, this is wrong.
- E.g., consider: `findall(X, p(X), L)`
  - The called predicate `p` would normally be searched in the library module that defines `findall`.
  - However, the called predicate `p` is really defined in the module that calls `findall`.
- To solve this problem, `findall` is specially declared as a meta predicate in the library.

Meta Predicates (2)

- The following declaration is used:

```
:- meta_predicate findall(?, 0, -).
```

[https://www.swi-prolog.org/pldoc/doc/_SWI_/boot/bags.pl?show=src]

- The declaration states that in the second argument (the call), the current module prefix should be added to the functor before `findall` is called.

  The module prefix (of the context in the call to `findall`) is only added if there is no module prefix yet. The number "0" is used to indicate that there will be no additional arguments in the call. For other predicates, such as a general `map_list`, the passed term is not the final call, but will get more arguments (the input and output list elements). In that example, 2 would be used. The other arguments "?" and "-" in the meta predicate declaration are normal mode indicators (not module sensitive).
  [https://www.swi-prolog.org/pldoc/man?section=metapred]

Operators

- In SWI Prolog, very module has its own operator table.
- Each operator table is initialized with the operator table of the "`user`" module.
- One can declare operators in the export list of a module declaration:

```
:- module(relatives,
          [op(700, xfx, father_of),
           father_of/2,...]).
```

- When a module uses the module "`relatives`", the exported operators are defined in the using module.

  If one explicitly lists the imports, one can use e.g. `op(_,_,father_of)`.

- Import/Export of operators is not portable (only to YAP).

References

- SWI Prolog Reference: 6. Modules
  [https://www.swi-prolog.org/pldoc/man?section=modules]
- G. Kniesel: Advanced Logic Programming, Chapter 6: "Modules"
  [https://sewiki.iai.uni-bonn.de/_media/teaching/lectures/alp/2012/08_-_modules_and_objects.pdf]
- Michael T. Richter: Using SWI-Prolog's modules.
  [https://chiselapp.com/user/ttmrichter/repository/gng/doc/trunk/output/tutorials/swiplmodtut.html]
- SICStus Prolog: The Module System
  [https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_7.html]
- Daniel Cabeza and Manuel Hermenegildo: A New Module System for Prolog.
  [http://oa.upm.es/14635/1/HERME_REFWORKS_1999-1.pdf]
- INCITS/ISO/IEC 13211-2:2000[S2016]: Information technology - Programming languages - Prolog - Part 2: Modules