ISO/IEC DTR 13211–2
Modules: Amendment
N251

Editor: Jonathan Hodgson
jhodgson@sju.edu

November 1, 2013

Introduction

This amendment to ISO/IEC 13211-2 Modules serves to clarify the requirements for a standard conforming implementation of Prolog modules. In particular it addresses conformance issues left unspecified in clause 6 of ISO/IEC 13211-2.

A conforming module system is required to provide the following.

1. A method of specifying the prolog text defining a particular named module. (**??**).
2. A method to indicate which predicates are to be made available (exported) for potential use without module name qualification. (**??**).
3. A method of indicating which other modules are being used (imported) be a modules and whose exported predicates may thus be used without module qualification by the importing module. (**??**).
4. A method to indicate which predicates defined by a given module are context sensitive, and for such predicates which arguments are context sensitive. (**??**).

The clause numbering of this amendment follows that of ISO/IEC 13211-2.

6 Language Concepts and Semantics

6.2.4 Module directives

6.2.4.1 Module directive/2

The module directive `module(Name,List))` specifies that the Prolog text in which it appears forms the body of the module `Name` and that the predicates whose predicate indicators appear in the list `List` are exported by the module.

6.2.5 Module body

A Prolog text that begins with the module directive `module/2` (??) defines the body of the module named in the directive. The module body terminates with the end of the Prolog text.

6.2.5.5 Directive use_module/2

A directive `use_module(F,L)` in the body of a module `M` where `F` is the name of a Prolog text and `L` is a list of predicate indicators identifying a set of the predicates exported by the module determined by `F` specifies that the modules `M` imports the predicates in the list `L`.

6.2.5.6 Module directive use_module/1

A directive `use_module(F)` in the body of a module `M` where `F` is the name of a Prolog text specifies that the module Prolog text in which it appears imports all the predicates exported by the module determined by `F`.

NOTE – The directive `use_module(F)` in the body of a module `M` has the same effect as the `ensure_loaded(F)` in the sense that the public predicates of the module determined by `F` are imported into `M`.

6.2.5.7 Directive meta_predicate/1

A directive `meta_predicate MIS` in the body of a module `M` where `MIS` is a sequence of metapredicate mode indicators indicates that the predicates so identified are context sensitive and that the arguments whose modes are `:` must be prefixed with the current source module in which their activation takes place.

6.4 Context sensitive predicates

6.4.4 Examples: Metapredicates

6.4.4.3 Use of the module directive meta_predicate/1

This example illustrates the use of the directive `meta_predicate/1`

```
:- module(example, [a/1]).

:- meta_predicate a(:).

a(G) :- call(G).
```

When the exported predicate `a/1` is called in an importing module `M` the effect of `a(G)` is the predicate `call(example:G)` so that `G` is called in the context of `example` rather than in the context of `M`.
The following example illustrates the use of a metapredicate to obtain context context information for debugging purposes.
Suppose that the following Prolog text is indicated by the name `trace`:

```
:- module(trace, [tr/1]).

:- meta_predicate tr(:).

tr(Goal) :-
      Goal = Module : G,
      inform_user('CALL', Module, G),
      call(Goal),
      inform_user('EXIT', Module, G).
tr(Goal) :-
      Goal = Module : G,
      inform_user('FAIL', Module, G),
      fail.
inform_user(Port, Module, Goal) :-
      write(Port), write(' '), write(Module),
      write(' calls '), writeq(Goal), nl.
```

and that the following Prolog text is indicated by the name `foo`;

```
:- module(foo, [a/1]).
:- use_module(trace).

a(X) :- tr( b(X)).

b(7).
```

Assuming that the directives `use_module(trace)` and `use_module(foo)` are in effect.

The goal: tr(A(X)) succeeds writing:

```
CALL user calls a(_131111)
CALL foo calls b(_131111)
EXIT foo calls b(7)
EXIT user calls a(7)
```

The following example also illustrates the use of the module `trace`.
Suppose that the Prolog text identified by `badsort`contains:

```
:- module(sort_with_errors, [bad_sort/2]).

:- use_module(trace).



bad_sort(List, SortedList) :-
    bad_sort(List, SortedList, []).
bad_sort([], L,L).
bad_sort([X|L], R0, R) :-
     tr(split(X,L,L1,L2)),
     bad_sort(L1,R0,R1),
bad_sort(L2,[X|R1], R).
split(_, [],[],[]).
split(X, [Y|L], [Y|L1], L2) :-
      Y @<X, !,
      split(X,L, L1,L2).
split(X, [Y|L], [Y|L1], L2) :-
      split(X,L, L1,L2).
```

Assuming that the directives `use_module(trace)` and `use_module(badsort)` are in effect,
`bad_sort([3,2,1], L)`
fails writing

```
CALL sort_with_errors calls split(3,[2,1],_131158,_131159)
EXIT sort_with_errors calls split(3,[2,1],[2,1],[])
CALL sort_with_errors calls split(2,[1],_131170,_131171)
EXIT sort_with_errors calls split(2,[1],[1],[])
CALL sort_with_errors calls split(1,[],_131180,_131181)
EXIT sort_with_errors calls split(1,[],[],[])
FAIL sort_with_errors calls split(1,[],_131180,_131181)
FAIL sort_with_errors calls split(2,[1],_131170,_131171)
FAIL sort_with_errors calls split(3,[2,1],_131158,_131159)
```