A Preprocessor Based Module System for Prolog

Roland Dietrich
GMD Forschungsstelle an der Universität Karlsruhe
(German National Research Center for Computer Science)
Haid-und-Neu-Straße 7, D - 7500 Karlsruhe

Abstract

In this paper, we propose a simple module system for Prolog. A minimal set of simple concepts realize the most important objectives of a module system: structuring of a larger piece of software into smaller logical units, information hiding, and abstract data types. It can be completely implemented by a preprocessor which maps modularized Prolog programs onto ordinary Prolog programs. The preprocessor itself can be written in Prolog and thus the module system can be integrated in any existing Prolog environment. It can easily be integrated with other preprocessor based software engineering aids, for example static mode and type checkers.

1 Introduction

For developing and maintaining large software systems, a modularization principle is essential for any real programming language. Module systems comprise an important feature of modern imperative programming languages like Modula-2 [Wirth 83] or Ada [Ada 83], and also for high level declarative languages (e.g. [Goguen & Meseguer 84, MacQueen 84]). (For a general overview on module systems for programming languages see [Drosopoulou 88].) As efficient implementations of Prolog, for example based on the Warren Abstract Machine (WAM) [Warren 83] or on special hardware, become more and more available, and Prolog is more and more used even in industrial environments, the development of powerful module systems for this language is an important task for every prolog implementor. A module system is also planned to be a part of a forthcoming international Prolog standard [Scowen 88].

But until now, there is no common agreement within the Prolog community about how such a system should look, especially how certain problems arising from some dynamic facilities of Prolog, e.g. meta call and program modification, should be solved.

In this paper, we propose a very simple, but nevertheless powerful enough module system for Prolog. It can be competely implemented by a preprocessor which maps modularized Prolog programs onto ordinary Prolog programs. The preprocessor itself can be written in Prolog and thus the module system can be integrated in any existing Prolog environment.

In our module system the interface of a module identifies *global names*, which can be predicate or function names, and which are visible and usable in any environment *importing* the module. All other names are local to the module and invisible and unusable to other modules. That is, the only information a programmer of a module has to give are the predicates and functions which the module implements and provides to an external user, and the modules whose exported predicates he wants to use. Explicit hierarchical structuring of modules is not possible (and not neccesary, because one can model hierarchical structures by means of import dependencies in a flat module system). These simple and minimal concepts realize the most important objectives of a module system: structuring of a larger piece of software into smaller logical units, information hiding, and abstract data types in the sense that the interface of a module declares a collection of data (global functions for building terms) and a collection of operations on these data (global predicates). The operations are defined or implemented by the respective clauses of the module's program.[1] Furthermore, the module system can be a basis for a library of Prolog programs.

The main principle of our implementation is that all local names of a module will be internally renamed and made anonymous to all other modules, and all global names of a module, i.e. exported and imported names, will be automatically qualified. This also restricts the often cumbersome need for explicit qualification of names imported from other modules to the case, when a name is imported more than once from different modules.

Our module system can easily be integrated with other preprocessor based software engineering aids, for example static mode and type checkers [Mycroft & O'Keefe 84, Dietrich 88, Dietrich & Hagl 88]. Especially, the integration with a static type system will improve the facilities to define abstract data types, because it enables data types and operations to be explicitely related by signatures.

In the next section we first describe the module system from the programmer's point of view, that is we describe the syntax and (informally) the semantics of a module. In section 3 we outline the implementation principles of the system and present the algorithm the preprocessor is based upon. In section 4, we summarize our results, give some references to related work and identify some future work. We assume familiarity of the reader with the Prolog language and concepts as described in [Clocksin & Mellish 81].

2. The Module System

In this section we show the concepts of our module system from a programmer's point of view. That is we define the abstract structure of a module, show a possible syntax, and explain the semantics. We suppose that there is "some Prolog system", which implements the language described in [Clocksin & Mellish 81]. But the module system can be adapted to an arbitrary Prolog system.

2.1 Modules

Definition 1 *(Structure of Modules).*

- A *module M* consists of a module name, an interface and a program.

---

1 In algebraically defined abstract data types the operations are also functions and specified by equations.

- The *interface* of M consists of an import list, a set of operator definitions, a predicate list, and a function list. All of them may also be empty.
  - The *import list* of the interface of a module M contains module names. These are the modules whose exported predicates and functions are intended to be used within M.
  - An *operator definition* defines dynamically a new operator for constructing terms of prolog programs. The definition identifies the name of the operator, how it is to be used (infix, post-fix, prefix), and its precedence. For example, in [Clocksin & Mellish 81] the system predicate *op(_,_,_)* is for that purpose. All operators defined in the interface of M should also be contained in the function or predicate list. (Otherwise they are local and should be defined in the program of M)
  - The *function list* of a module interface contains all function names with arity wich are exported and made usable in modules importing this module.
  - The *predicate list* of a module interface contains all predicate names with arity wich are exported and made usable in modules importing this module.
- The *program* of a module is a set of Prolog-clauses (facts, rules, and goals).

■

Figure 1 schows the structure of a module in a BNF-like form. The nonterminal *<clause>* denotes a prolog clause, *<atom>* a Prolog atom (i.e. a name which is not further decomposable). We also adapt a special way of specifying the arity of a function or predicate symbol: for example the 3-ary function f is denoted f(_,_,_). This notation has the advantage that implicitly the usage of an operator is also specified, e.g "_+_".

Figure 2 schows a module *lists* specifying some list processing functions[1], and the interface of module *arithmetic*, specifying some part of the integer arithmetic which usually is built-in in Prolog implementations. Note that the implementation of the *reverse* predicate uses difference lists. The predicate *dl_rev* and the operator "&" are local. The functions and predicates exported from *arithmetic* are supposed to be "built-in". Besides the names in the function and predicate lists, we assume that all numbers are implicitly member of the function list of *arithmetic*. The operator definitions are those of CProlog [Pereira 84].

2.2 *Name Classes*

Within a module one can distinguish different kind of (predicate, function or operator) names, depending whether they occur in the interface of the module or not, or in the interface of an imported module.

Definition 2 (*Name Classes*).
Let n be a function or predicate name (i.e. a Prolog atom including an arity) occuring in a module M.

- n is a *exported name* of M iff n occurs in the function list or in the predicate list of M.
- n is an *imported name* iff there is a module M' occuring in the import list of M such that n occurs in the function or predicate list of M'.

---

1 We use here the somewhat awkward notation with explicit "cons" function. In practice, of course, the system would provide the more convenient [... | ...] - notation.

```
<module>       ::=  "module" <atom> "."
                    <interface>
                    <clause>*
                    "end" <atom> "." .

<interface>    ::=  "import" <atom_list> "."
                    <operator_def>*
                    "functions" <arity_list> "."
                    "predicates" <arity_list> "." .

<atom_list>    ::=  "[]" .
<atom_list>    ::=  "[" <atom> "|" <atom_list> "]" .

<arity_list>   ::=  "[]" .
<arity_list>   ::=  "[" <arity_spec> "|" <arity_list> "]" .

<arity_spec>   ::=  "_" <atom> "_" .              % infix-operator
<arity_spec>   ::=  <atom> "_" .                  % prefix-operator
<arity_spec>   ::=  "_" <atom> .                  % postfix-operator
<arity_spec>   ::=  <atom> .                      % constant
<arity_spec>   ::=  <atom> "(" <arity> ")" .      % n-ary functor

<arity>        ::=  "_" .
<arity>        ::=  "_" "," <arity> .

<atom>, <operator_def>, and <clause> are defined by the Prolog system.
```

Figure 1: Syntax of a module.

```
module lists.

import [ arithmetic ].

predicates [ append(_,_,_) , reverse(_,_), length(_,_), member(_,_) ].

functions [ nil , cons(_,_) ].

append(nil,L,L).
append(cons(X,L1),L2,cons(X,L3)) :- append(L1,L2,L3).

:- op(500,yfx,&) % local operator for difference lists

reverse(L,R) :- dl_rev(L,R & []).

dl_rev([X|L],R & S) :- dl_rev(L,R & [X | S ]).
dl_rev([],L & L).

length([],0).
length([X|L],N) :- length(L,N1), N is N1 + 1.

member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

end lists.

module arithmetic.

:- op(500,yfx,[+,-]).    % binary +,-
:- op(500,fx,[+,-]).     % unary +,-
:- op(400,yfx,[*,/]).

functions [ _ + _ , _ - _ , + _ , - _ , _ * _ , _ / _ , exp(_), log(_) ].
                                              % and 0, 1, 2, 3, 4, ...

predicates [ _ is _ , integer(_) ].

% program is built in

end arithmetic.
```

Figure 2: Modules for list processing and integer arithmetic.

- n is a *local name* iff it is neither an exported nor an imported name.
- n is a *defined name* of M, iff n is a predicate name and there is a clause in the program of M such that n is the principal functor of the clause's head (i.e. the clause has the form *p(...) :- body.*).

Now let n be any function or predicate name, not necessarily occuring in M.

- n is a *transported name* or *indirectly imported name* of M iff there is a module M' occuring in the import list of M such that n is either an imported name of M' or a transported name of M'.

Let in the following *exported(M)*, *imported(M)*, *local(M)*, *transported(M)*, and *defined(M)* denote the set of exported, imported, local, transported, and defined names of M, respectively.

■

The transported names of a module are not important from the programmers point of view, but the implementation must take care, that transported names cause no name confusions when binding several modules (c.f. section 3.).

Figure 3 shows a module hierarchy. It can be implemented by means of our flat module system as schown in Figure 4. Besides the import lists, which define the hierarchy, the interfaces contain some

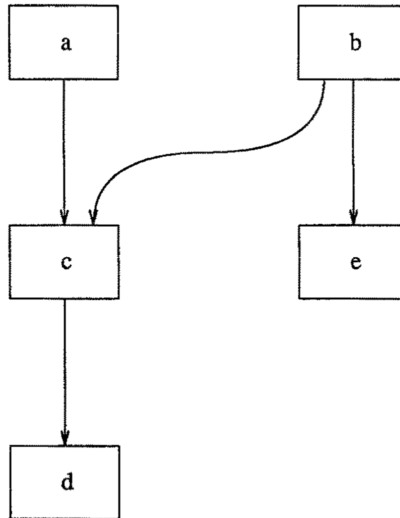

Figure 3: A module hierarchy

example function and predicate lists, the programs some example clauses. With respect to the programs of Figure 4 we have the following name classes:

	a	b	c	d
exported	fa, pa	fb, pb, q	fc, pc	q
imported	fc, pc	fc, pc, r	q	
local	q			
transported	q	q		
defined	pa, q	pb,q	pc	q

2.3 Name Conflicts

Because we want to avoid extensive qualification of names with their module membership as far as possible, to some extent the programmer of a module has to take care, that every name belongs to a unique module. Especially, that the definition of every predicate can be associated with a unique module. That is, for example, the programmer has to chose the exported and local names of his module different from all imported names. This is expressed in the following conditions for modules:

```
module a.

import [ c ].
functions [ fa(_) ].
predicates [ pa(_) ].

pa(fa(X)) :- q, pc(X).
q.

end a.
```

```
module b.

import [ c , e ].
functions [ fb(_) ].
predicates [ pb(_), q ].

pb(fb(X)) :- q, pc(X), r.
q.

end b.
```

```
module c.

import [ d ].
functions [ fc(_) ].
predicates [ pc(_) ].

pc(fc(z)) :- q.

end c.
```

```
module d.

import [].
functions [ ].
predicates [ q ].

q.

end a.
```

```
module e.

import [].
functions [ ].
predicates [ r ].

r.

end b.
```

Figure 4: Implementation of the module hierarchy of Figure 3

Definition 3 *(Conflict Freeness of Modules).*
Let M be a module.

M is said to be *basically conflict free* iff for any two modules M_1 and M_2 occuring in the import list of M the following conditions hold:

(CF1) $exported(M) \cap imported(M) = \varnothing$

(CF2) $exported(M_1) \cap exported(M_2) = \varnothing$

M is said to be *conflict free* iff it is basically conflict free and in addition for any two modules M_1 and M_2 occuring in the import list of M the following conditions hold:

(CF3) $transported(M) \cap exported(M) = \varnothing$

(CF4) $transported(M_1) \cap transported(M_2) = \varnothing$

■

Basic conflict freeness expresses, that the names of the direct environment of a module cause no name conflicts. General conflict freeness also regards names of indirectly imported modules (that is it takes into account the transitive closure of the import relation.)

Perhaps one would expect also a condition like $exported(M) \cap local(M) = \varnothing$ or $imported(M) \cap local(M) = \varnothing$. But note that by definition any exported or imported name definitively cannot be local (cf. Definition 2). That is, these conditions are trivially true.

Because a programmer has to *determine* the exported names of a module and he *knows* the imported names (by inspection of the interfaces of imported modules), it is reasonable to expect that he is able to ensure the basic conflict freeness of modules (conditions (CF1) and (CF2)). (CF1) can easily be avoided by choosing the right names. If there are name conflicts with respect to (CF2), the program-

mer must resolve them by appropriate qualification. On the other hand, he does not necessarily know the transported names. Therefore, if (CF3) and (CF4) are violated, uniqueness of names must be ensured by the implementation of the module system.

The correct qualification of ambiguous names is expressed by the following

Definition 4 (*Qualification rule*).
Let M be a module. Let M_1 and M_2 be modules occuring in the import list of M, and $n \in exported(M_1) \cap exported(M_2)$. Within M, the atoms $M_1{:}n$ and $M_2{:}n$ are called *qualified names* and denote the name n which is exported from M_1 and M_2, respectively[1]. It is not allowed to use the name n unqualified within M.

■

Besides name conflicts, there are three other important properties which must be kept by a module, namely that every exported predicate is defined, that no imported predicate is redefined, and that an operator defined in the interface of a module is to be exported:

Definition 5 *(Well Defined Interface)*.
Let M be a module and $operators(M)$ the set of operators defined in the interface of M. We say that the interface of M is *well defined* iff the following conditions hold:

(WD1) $exported(M) \subseteq defined(M)$

(WD2) $imported(M) \cap defined(M) = \emptyset$

(WD3) $operators(M) \subseteq exported(M) = \emptyset$

■

2.4 Built-in Modules

In order to deal with buil-in predicates and data (like numbers), there are principally two different approaches: we can treat every built-in predicate and function symbol (e.g. 1,2,3,... or "[_|_]") as global names, which need not to be imported explicitly, and which are not allowed to be redefined in other modules. That is, we have *one* virtual system module which is implicitly imported by all other modules. Or, as suggested in Figure 2 for arithmetic, we can identify several groups of built-in features which belong to *different* virtual system modules which must explicitly be imported if needed. Again, names imported from system modules are not allowed to be redefined and built-in names cannot be used qualified, because the implementation of these modules is part of the underlying Prolog implemetation and does not recognize qualified names.

3. The Implementation

In this section we describe the implementation of the module system by means of a preprocessor which maps modules onto ordinary Prolog programs.

1 Note that this rule needs an appropriate modification of the syntax of programs, specifying a syntactic object "qualified name".

The principle of our implementation is very simple and can be summarized as follows: every exported and imported name of a module is qualified by the corresponding module name, and every local name obtains a prefix which is unknown to every other module. Qualification is done automatically in a consistent way in the modules where a name is defined and exported, and in the modules where it is imported and used.

3.1 Processing Modules

The following algorithm *process(M,E)* describes the global behaviour of the preprocessor for modules. The input to this algorithm is a module M. The output is the set E of exported names of M. As a side effect, the algorithm outputs the modified program of M to a well defined file. The modification of the program is according to the above explained principle. The algorithm is called recursively with all modules occuring in the import list of the interface of M. That is, after the algorithm has terminated, the output file containes the modified programs of M and all modules which are directly or indirectly imported by M.

The algorithm uses the following elementary functions and notations, whose implementation is not further detailed here:

- $M.i$ and $M.p$ denote the interface and the program of a module M, respectively.
- *operators* (I), *imports* (I), and *exports* (I), denote the set of operator definitions, imported modules, and exported names specified in the interface I.
- *qualified* (n,M) is true, if n a is a name qualified with module M, i.e. n is of the Form $n:M$.
- *qualify* (n,M) adds a prefix "M:" to the name n.
- *rename* (n,M) adds a prefix to the name n. The prefix is well defined with respect to the module name M, but anonymous to the "outside world" of M.
- *output* (T) outputs a term T (e.g. a Prolog clause or operator definition) to a well defined File.

preprocess(M,E);

Input: a module M.
Output: the set E of exported names of M, associated with the module name M $((n,M) \in E)$.

% G is the set of names which are global in M, associated with a module name.
$G := \emptyset$;

% CS is the "conflict set", the set of names which are imported more than once.
$CS := \emptyset$;

% first preprocess imported modules and detect (CF2) conflicts
<u>for</u> each $M' \in$ *imports* $(M.i)$ <u>do</u>
    preprocess(M',E');
    <u>for</u> each $(n,M) \in E'$ <u>do</u>
        <u>if</u> there is M' such that $(n,M') \in G$
        <u>then</u> $CS := CS \cup \{n\}$
        <u>fi</u>
    <u>od</u>
    $G := G \cup E'$

```
od;
% compute exported names and operators and detect (CF1) conflicts
E := ∅;
for each n ∈ exports(M) do
    if there is M' such that (n,M') ∈ G then CS := CS ∪ {n} fi;
    G := G ∪ {(n,M)}; E := E ∪ {(n,M)}
od.
Ops := operators(M.i);
P := Ops ∪ M.p;
transform(P,G,CS)
end preprocess.
```

transform(P,G,CS);

Input: A program P, a set of names G associated with modules (global names), and a set of names CS (conflict set);

```
for each clause c ∈ P do
    for each name n occuring in c do
        if qualified(n,M)   % conflict resolved by the programmer
        then
            if (n,M) G
            then error(''illegal qualification'',M:n)   % else n is left unchanged
            fi
        else
            if n ∈ CS   % (CF1) or (CF2) conflict
            then error(''ambiguous name'',n)
            else
                if (n,M) ∈ G
                then qualify(n,M)   % ensure (CF3) and (CF4)
                else rename(n,M)   % hide a local name
                fi
            fi
        fi
    od
    output(c)   % output the transformed clause
od;
end transform.
```

The algorithm *preprocess* also determines the order in which the programs of dependend modules are output. For example preprocessing the module a of Figure 3, the respective programs are output in order d,c,a.

Figure 5 shows the result of preprocessing the module a of Figure 4 and Figure 6 shows the result of preprocessing module b of Figure 4 (cf. section 2). The renaming function for local names takes as prefix 'xy' followed by the module name and '#' for readability reasons. In the final implementation of the system, the prefix will be anonymous and not directly related to the module names in order to

```
% clauses from module d
d:q.

% clauses from module c
c:pc(c:fc(z)) :- d:q.

% clauses from module a
a:pa(a:fa(X)) :- xya#q, c:pc(X).
xya#q.
```

Figure 5: Preprocessing module a.

```
% clauses from module d
d:q.

% clauses from module c
c:pc(c:fc(z)) :- d:q.

% clauses from module e
e:r.

% clauses from module b
b:pb(b:fb(X)) :- b:q, c:pc(X), d:r(X).
b:q.
```

Figure 6: Preprocessing module b.

prevent illegal use of local names at runtime.

The algorithm *preprocess* ensures conflict freeness of modules in the following way:

- (CF1) and (CF2) should be guaranteed by the programmer. If one of them is hurt, an error message is produced.
- (CF3) and (CF4) is enforced by qualification of exported *and* imported names with their module name. This is done at every occurence of a name within the program.

It is easy to extend the algorithm to check also the well-definedness of interfaces (cf. Definition 5, section 2), to deal with built-in predicates and modules (cf. section 2.4), and to detect cyclic dependencies between modules.

3.2 Processing Goals

Now suppose that we have preprocessed one or several modules (and thus all directly and indirectly imported modules). This has resulted in one ordinary Prolog program P implementing the collection and interaction of these modules. In order to execute goals with respect to this program, one must identify a module context M, wherein it is to be executed (e.g. via a built-in predicate *execute(Goal,M)*). Then the the clause *Goal* is transformed with respect to this module context like an ordinary clause by the algorithm *preprocess*. That is, names occuring in the goal are qualified with the respective module name if they are exported from M or imported from an other module, and otherwise they are renamed like local names of M. Finally the transformed goal is executed with respect to the program P. For this purpose, it is necessary to keep the information about all modules' interfaces which have been preprocessed before.

An other possibility is to write goals directly into the program of a module (which is possible in Prolog, anyway). This module (and all imported modules) can be preprocessed as described in section 3.1. The goals usually are automatically executed when loading the preprocesed program into an interpreter (''consult''), or when executing the compiled version. The exact procedure depends on the underlying Prolog implementation.

3.3 Behaviour at Runtime

Prolog has several dynamic features which may cause some problems for module concepts. Especially the analogy between data and programs, which allow dynamic construction of calls and even program modification. Some module systems, e.g. the one of KA-Prolog [Goos et. al. 87, Lindenberg et. al. 87] simply forbid some of the critical features, e. g. modification of the program database, and allow meta calls only within an explicitly defined module context. But this restricts some of these features which have proven to be useful in many typical applications of Prolog. Imposing restrictions on the use of Prolog features would always decrease the excellent rapid prototyping qualities of Prolog, which we want to preserve also within a module system.

Therefore, we do not impose any restrictions on the use of Prolog within modules, and argue that the semantics of every piece of program or goal is uniquely defined by our preprocessing algorithm. We demonstrate this by means of an example (it is taken from [Ultsch et al. 88]). Figure 7 shows two modules *a* and *b* and their preprocessed version.

```
module a.                  module b.              % module a preprocessd:

export [ f(_) ].           import [ a ].          xya#g :- true.
                                                  a:f(X) :- call(X).
g :- true.                 g :- fail.
f(X) :- call(X).                                  % module b preprocessed:

                                                  xyb#g :- fail.
```

Figure 7: Metacall and Modules.

The question now is whether the goal

```
:- execute(f(g),b).
```

succeeds or fails. Or, in other words, does the argument "*g*" of the call of "*f*" belong to module *a* or module *b*, when called in the context of *b*? The goal is transformed as outlined in section 3.2 into the following goal

```
:- a:f( xyb#g )
```

and this goal fails with respect to the preprocessed program.

4. Conclusions

We have presented a simple module system for Prolog which allows structuring of large software systems, information hiding and the implementation of abstract data types. Within a module interface one can (1) declare operators, (2) specify predicates and functions which are exported, and (3) modules, which are imported and whose exported predicates can be used. Our module system is thus *name based* like those of MProlog and Micro-Prolog [Szeredi 82]. Other module systems are only predicate based, that is only predicate names can be exported and all function names are considered to be global, e.g. ECRC-Prolog [Chan & Poterie 87] and KA-Prolog [Goos et. al. 87, Lindenberg et. al. 87]. It is easy to realize a predicate based module system in the same manner as shown here, if we leave out function lists from the syntax of interfaces and change the preprocessing algorithm in an obvious way.

Most module concepts require that every imported name is explicitly imported and do not allow one to import the same name from different modules. Our minimal export/import and qualification concept alows the import of *complete* modules, i.e. every exported name of this module is available. The need for qualification is restricted to the cases where a name is imported twice. Thus gaining more flexibility (no restriction on imports) without introducing too much notational overhead (no explicit import of names, only minimal need for qualification). The uniqueness of the names of a module can be decided at preprocessing time.

Modularized programs can be transformed in a systematic way onto ordinary Prolog programs, maintaining the module structure by means of appropriate prefixing of names. This transformation process defines a unique behaviour of modules at runtime, also for the dynamic features of Prolog like meta call and program modification. Therefore, in contrast to other module systems, we do not need to restrict the use of any of these features within modules. It is our philosophy to provide the means for safer programming of larger software by several people, but on the other hand not to restrict other qualities of Prolog which, for example, support rapid prototyping.

A major advantage of our preprocessing approach is that it is independent from a certain Prolog implementation and can be adapted to any Prolog system, whether compiler or interpreter based. Furthermore, modules can be implemented in other languages than Prolog, as soon as the underlying Prolog system provides an interface to these languages. Because all global names are prefixed by the module name after transformation, one can use an ordinary Prolog debugger for debugging transformed modules. In order to enable separate transformation of modules and goals and oinding of transformed modules, the implementation must be extended by some kind of module management system, which stores, besides the transformed programs, information about the original interface of the modules.

It is planned that the module system will be integrated with a mode and type system for Prolog [Dietrich 88, Dietrich & Hagl 88], which is also realized as a preprocessor. Then the interface of a module will also allow the specification of types, and the declaration of types and input/output modes for arguments of predicates (i.e. signatures). This will further improve the facilities to specify and implement abstract data types, and the programming security.

Acknowledgements. I would like to thank my colleagues from GMD Karlsruhe, especially H. Lock, P. Kursawe, and F.-W. Schröer, for many fruitful discussions and useful comments on earlier versions of this paper.

References

The Programming Language Ada, Reference Manual, American National Standards Institute, Inc., ANSI/MIL-STD-1815A-1983, Springer LNCS 155, 1983.

Y. C. Chan, B. Poterie, *Modules in Prolog*, British Standards Institution - IST/5/15 Prolog, Document PS/185, February 1987.

W. F. Clocksin, C. S. Mellish, *Programming in Prolog*, Springer Verlag Heidelberg, 1981.

R. Dietrich, *Modes and Types for Prolog*, Arbeitspapiere der GMD Nr. 185, February 1988.

R. Dietrich, F. Hagl, *A Polymorphic Type System with Subtypes for Prolog*, Proc. 2nd European Symposium on Programming (ESOP '88), Nancy, March 1988, Springer LNCS 300, 79-93.

S. Drosopoulou, *Module and Type Systems - a Tour*, Imperial College, London, Report IC/FPR/PROG/2.2/12, issue 2, March 1988.

J. A. Goguen, J. Meseguer, *Equality, Types, Modules, and Generics for Logic Programming*, Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, 1984, 115-126.

G. Goos, R. Dietrich, P. Kursawe, *Prolog Arbeiten in Karlsruhe*, in: W. Brauer, W. Wahlster (Eds.), *Wissensbasierte Systeme*, Springer Informatik Fachberichte 155, September 1987, pp 89 - 104.

N. Lindenberg, A. Bockmayr, R. Dietrich, P. Kursawe, B. Neidecker, C. Scharnhorst, I. Varsek, *KA-Prolog: Sprachdefinition*, Universität Karlsruhe, Interner Bericht 5/87 und Arbeitspapiere der GMD Nr. 249., May 1987.

D. MacQueen, *Modules for Standard ML*, ACM Symposium on LISP and Functional Programming, Austin, Texas, 1984.

A. Mycroft, R. A. O'Keefe, *A Polymorphic Type System for Prolog*, Artificial Intelligence 23, 1984, 295-307.

F. Pereira (Ed.), *C-Prolog User's Manual*, Version 1.5, EdCAAD, University of Edinburgh, February 1984.

R. S. Scowen (Ed.), *Modules in Prolog - A Discussion Paper*, ISO/IEC JTC1 SC22 WG12, Document N14, July 1988.

P. Szeredi, *Module Concepts for Prolog*, Proc. Workshop on Prolog Programming Environments, Linköping, 1982, pp. 69-80.

A. Ultsch, M. P. J. Fromherz, H.-P. Schmid, *Modules in Prolog*. In: R. Scowen (Ed.), *PROLOG - Oxford 1988 papers*, ISO/IEC JTC1 SC22 WG17, Document N12, April 1988.

D. H. D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, Artificial Intelligence Center, SRI International, 1983.

N. Wirth, *Programming in Modula-2*, Springer Verlag, 1983.