TCPW/CW/K.INT.

BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

Modules in Prolog, once more

by
Gerda JANSSENS **

Internal Report
BIM-prolog IR4

May 1984

* BIM
Kwikstraat 4
B-3Ø78 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 2ØØA
B-3Ø3Ø Heverlee Belgium
tel. +32 16 2Ø Ø6 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

MODULES IN PROLOG, ONCE MORE
(*)

G.Janssens

Departement Computerwetenschappen K.U.Leuven
Celestijnenlaan 200A
B 3030 Heverlee Belgium
016/20.06.56

Abstract
We briefly discuss the modular programming technique as used in the classic procedural programming languages and we review the currently known PROLOG module models.
We explain the concept of our proposal and the interface declarations introduced to support the user. We compare our facilities with the ones provided in a language as ADA (ADA is a trademark of the US Departement of Defense). Finally a detailed critique of other approaches to introduce modules in PROLOG is given.

Program Area : Logic Programming Languages

1. Modular programming

1.1. Motivation

It really is a burden when support for modularity is lacking in a programming language, e.g. when one wants to build libraries or wants to provide abstract data types. Such techniques are currently used in large software development projects.
In the case of PROLOG, the absence of local names often results in the inadvertly use of the same name and is a major source of errors in developing large programs.

1.2. Modules in procedural programming languages

Parnas's technique [Parn] for module specification is the precursor of the syntax for abstract data types in the recent procedural programming languages such as ADA [Bern], CLU [Lisk] and Alphard [Shaw].

Parnas regards a module as the manager of a given (abstract) object. The subroutine calls to the module are regarded as functions that observe or alter states of the object.
The declaration provides the user and the implementer with all (and only) the necessary information. The details of implementing the objects are hidden in the module and can be changed at any time, without any consequences for the user, as long as they are conform to the declaration.

In recent programming languages such as ADA it is also possible :

1 to specify the external properties of the abstract data type by a predefined formalism (declaration).
In Alphard the declaration includes the specification of the effect of the procedures, providing an extra facility to verify programs.

2 to define the declaration and the implementation as separate compilation units.

(*) This work is sponsered by the "Diensten voor de Programmatie van het Wetenschapsbeleid" under contract nr KBAR/SOFT/1

3 to use private types so that no structural details can ever be known by a user.

4 to restrict operations available on the objects of the type.

5 to use an exception mechanism for error handling.

6 to adapt easily an abstract data type to different situations quite similar to one that has motivated its creation : the generic units

The main objective of our proposal is to group together related procedures and data so that the minimum amount of essential information is distributed around the program.
While discussing our module mechanism we will verify which of these facilities are inherent to PROLOG and which of them are useful to provide abstract data types in PROLOG. This is important because PROLOG as a logic programming language has specific properties.

1.3. Other PROLOG models

Our objection to the existing proposals is that although they are all based on the same idea, namely the support for modularity, they interpret the concept on their own specific way. The lack of a simple conceptual model creates a confusing situation for the user.
Most of the models support the user insufficiently when he makes a typing error. The impact of an erroneous name depends on the kind of model.

2. Our solution

2.1. Concept of the solution

The concept is quite simple : in the PROLOG programs we use unambiguous names formed by prefixing the original names by the name of the module in which they are defined.
The unambiguous name of the object p in module m is m$p.
The prefixing is not necessary for names referring to the outside world, namely for data.

In fact, our approach consists only of a discipline of writing programs, it does not change anything to the runtime structures of PROLOG. However, to reduce the burden for the programmer, some syntactic sugar is provided. This syntactic sugar can be handled by a preprocessor and produces PROLOG as sketched in the above concept. Thus all programs can be mapped to this very simple conceptual model.

2.2. More in detail

We will describe how to obtain a more user friendly solution and how to provide support for the use of modules.

2.2.1. How to work with modules in PROLOG

1) We eliminate most of the boring prefixing by adding an interface declaration to the module. This declaration is nothing else than syntactic sugar.
If p, then q and r are the names of objects to de used in a module m0, the following interface declaration tells us that p is defined in module m1 and thus must be imported from m1, that

q is a global object and that r is a local one.

```
module m0
import p(..,..) from m1
global q(..)
local r
```

We propose as default options :

- for constants : global

- for other objects (functions and procedures) : local

2) A limited number of procedures may be used outside the module they are defined in. Therefore we introduce the notion of export information as a form of redundancy. This export declarations enumerate the procedures defined in the module which are allowed to be used by other modules

3) In the case of ambiguity the user himself must prefix the names by the relevant module ; it is allowed to have the following import declarations :

```
import empty() from stack
import empty() from queue
```

but the user has to distinguish between both by explicitly writing the prefix, i.e. stack$empty or queue$empty.
A procedure is normally characterized in the declarations by its name. Sometimes it is necessary to denote a procedure by the tupple name/arity to avoid confusion.

2.2.2. Support and safety

A preprocessor converts a module into the notation as sketched in section 2.1.
The default options do not cause any problems :

- a constant which is not explicitly declared, is a global object.

- a procedure which is not explicitly declared, is a local object.

The expansion is straightforward, different cases are sketched in the following example.

```
module m0
import p(..,..) from m1
global q(..)
local r

p(..,..)    is expanded into m1$p(..,..)
a           a constant is by default global and remains a
s(..,..)    is by default local and is expanded into m0$s(..,..)
q(..)       the default (local) is overridden : remains q(..)
```

r the default of a constant (global) is overridden : becomes mO$r

Moreover, the preprocessor can verify the consistency between the declarations and implementation of the module. The preprocessor verifies if all the procedures in the export declarations are defined in the module and if there does not exist any ambiguity within the module.

Referring to our comparison with the procedural languages we find that the first two facilities are not relevant for PROLOG.
The nature of the programming language PROLOG implies that our proposed formalism for interface declaration gives satisfaction : we have no types and so we do not need explicit descriptions of data types. In the case of procedures we have neither type restrictions for the parameters, nor input/output patterns to be satisfied.

Although we have no explicit data type description, the structural details can never be properly protected because of the availability of predicates as "clause".

It is up to the user wether he decides to restrict himself to the operations available on the objects of a given type. Modules can be combined in large PROLOG programs. Also here some verification is possible. The consistency between the import and export declarations can be checked. This verification cannot prevent the user from calling local predicates from other modules. Using build-in predicates as "clause", it is possible to obtain the (encoded) name of all local predicates (e.g. a metainterpreter providing a trace of the execution). Once the names are known, the metacall mechanism can be used to execute them. Preventing such a use is not possible without changing the meaning of "clause".

With our mechanism we can not write an exception handler based on propagation of raised exceptions. The import/export declarations are static and can not be context dependent.

As we have no types in PROLOG, all the modules are in fact generic units. We create data structures whose component values are unknown at the time of creation.

3. Detailed critic of other module models

3.1. MProlog [MPR], [Sze]

The complex interface declaration is implementation oriented. All the implementation dependent details are to be understood by the user and are not relevant to the module concept.
The control and support are comparable with ours.

3.2. Prolog-II [VCaneg]

The unambiguity of names is realized by characterizing objects by the tupple "(world,identificator)".
Every object belongs to the module or to the world where its name appears for the first time.
Each module is a node in a tree structure. All the objects of a given module are visible for the sons in the tree and are

invisible for the father and the brothers.
In this model you can not have a procedure empty for a stack and a procedure empty for a queue at the same time.
A mistyped name is interpreted as an object of a world closer to the root if that name appears there, and otherwise as a new object of the current world.

3.3. Proposal of Feuer [Feu]

He proposes a mechanism of complex names by prefixing with the module name.
In every module you specify the procedures to be exported. All the visible modules belong to the "view" of the module.
When you refer to the first procedure in the view with a specific simple name, you do not have to prefix it.
All the nonlocal names of a module in the view are accessible and no selection can be made.
A mistyped name can match an object name in the view.

3.4. Micro Prolog [McCabe]

In the current module the objects that are available are these appearing in the import/export lists as well as the local names.
We suppose that only one procedure with a specific name can be available at a time, because you can not specify the module the object belongs to. The manual is very unclear about the module facilities. We assume that the user himself must organize the module handling to avoid name clashes.

3.5. This proposal

Effects of mispelling the name of a procedure are considered local. An unintentional match is only possible with an explicitly imported name, otherwise a definition is missing, causing a compilation error.

4. Future research

It seems interesting to integrate the module concept in a special editor : this can further reduce the overhead for the user.
The PROLOG runtime environment can be adapted to the modular context and can provide some facilities to test modules (Again by removing the burden of explicitly prefixing names).
Program optimisation will be desirable for efficiency reasons : a great deal of the extra procedure calls due to the modular design can internally be eliminated.

Acknowledgements
I am grateful to M.Bruynooghe and Y.D.Willems for their suggestions and their helpful comments.

Bibliografie

[Bern]
L.Bernard, "The Handling of Abstract Data Types in ADA", Notes of a lecture at the Universite Libre de Bruxelles, may 1983

[Feu]
Alan Feuer,"Building Libraries in Prolog",
in the Proceedings of IJCAI-83, pp. 550-552

[Jones]
Simon B.Jones, "Structured Programming Techniques in Prolog",
in Logic Programming Workshop, 14-16 july 1980, pp. 322-333

[Lisk]
B.Liskov, A.Snijder, R.Atkinson, C.Schaffert,
"Abstraction Mechanism in CLU", in Communications of the ACM,
volume 20, number 8, august 1977, pp. 564-576

[McCabe]
F.G.McCabe, "Micro-Prolog Programmer's Reference Manual",
Logic Programming Associates Ltd., may 1981

[MPR]
"MPROLOG Language Reference Manual", version 1-2
Institute for Coordination of Computer Techniques in Hungary,
nov 1982

[Parn]
D.L.Parnas, "A Technique for Software Module Specification
with Examples", in Commmunications of the ACM,
volume 15, number 5, may 1972, pp. 330-336

[Shaw]
Mary Shaw, "Alphard : Form and Content",
Springer-Verlag New York, 1981

[Sze]
Peter Szeredi, "Module Concept For Prolog",
Institute for Coordination of Computer Techniques in Hungary,
in Proceedings of the Workshop on Prolog Programming
Environments (Sweden), pp.69-78

[VCaneg]
Michel Van Caneghem, "Prolog-II Manuel D'Utilisation",
Groupe Intelligence Artificielle Marseille, mars 1982