Perlog: A Prolog With Persistence and Modules

D. S. MOFFAT AND P. M. D. GRAY
Department of Computing Science, University of Aberdeen, Aberdeen AB9 2UB

The Perlog system is a Prolog system whose clauses and terms can persist on secondary storage as objects in a database. The unit of commitment is the module, and all atoms indirectly reference their module name, unless they are declared to be global. The module acts as a source of protection and reference to all its clauses, and also provides checks on import/export and on privacy, especially in the case of Abstract Data Types. This paper describes the structure of the module name space and the reasons for the decisions, particularly the use of metacalls to access procedures in other modules, and procedures with multiple definitions in different modules. A method for resolving global name clashes between previously committed modules is described. A model of the interpreter is presented in order to define the semantics of procedure call and assert and consult in a fully modular system. An example is given of a version of assert using B-tree routines in the database for clause storage. The system has been implemented in C and Prolog and PS-Algol under UNIX† on a VAX.‡

Received September 1987

1. INTRODUCTION

Prolog has shown itself to be a very good language for symbolic programming, but for large applications its single flat internal clause base proves to be a major drawback. In the Perlog system we have partitioned the clause base into modules and have incorporated an object store that enables the clause base to persist between sessions.

The object storage system is more general than Relational Database Management Systems and we can store general Prolog clauses as well as networks of objects, bitmaps and even frames with attached procedures of the language PS-Algol. Objects are retrieved from the object store incrementally and transparently. The Prolog modules are the units of commitment for the object store, and once modules have been copied to disc they may be shared between users and recombined in different configurations.

The modules are designed to support software development and to be as flexible as possible. A feature that distinguishes our modules from many others is that they can have local terms as well as the usual local procedures. In this respect our model of visibility is the same as that of MProlog[10] and Micro-Prolog,[6] but unlike them ours can resolve multiply defined procedures, treats both system and user predicates uniformly and has support for Abstract Data Types. The module design also aims to keep the semantics of the many meta-predicates, like call, clause, unit and assert, as close to the semantics that they have in a flat name space. Another system with modularisation is Multilog,[5] which supports hierarchies of worlds with three kinds of inheritance. We think that the semantics for meta-calls in Perlog are clearer and safer than those used in Multilog. Unlike the algebra for modules that O'Keefe[8] proposes, our system takes in hand the effect of clause and goal order in Prolog which has very practical consequences.

In the next section we re-argue for an *atom-based visibility* along the lines of Szeredi.[10] The section on meta-predicates affecting visibility discusses our new *resolve* and *of* predicates as well as *export* and *import*. Section 5 describes the features of the module system that are particular to a Prolog with persistence. As an example of the system in use we show how to have specialised storage within modules. Finally we give performance figures and draw some conclusions.

† trademark of AT & T
‡ trademark of Digital Equipment Corp.

2. VISIBILITY AS A PROPERTY OF ATOMS

As soon as we partition a large space, the objects within it attain the property of visibility which defines where they may now be seen. In terms of programming languages they have scope. In the module scheme that we propose for Prolog the modules are all at the same level, and hence objects have either local visibility – being visible only within one module – or they have global visibility and are visible to all modules. We restrict ourselves to a flat module space, but we can implement a hierarchical space on top of this by systematic renaming of predicates with a *resolve* predicate, as explained later.

Many other Prologs have a flat module space but they can be grouped into two categories based on their unit object of visibility, i.e. *predicate-based visibility*, in which only predicates have a visibility (DEC-10, ICL-Prolog, Quintus, Prolog-2, Waterloo Prolog) and *atom-based visibility* (MProlog, Micro-Prolog and our Perlog system), where an atom and all the functors and predicates that are built from it have the same visibility. In the atom-based model local terms cannot unify with other global terms nor with local terms from another module. Global terms have the same representation in all modules and may unify with other globals.

The first consequence of the atom-based visibility is that the visibility of procedures and terms is the same. This becomes important when we make use of a *meta-call* to call a procedure. Szeredi[10] proposes three principles for meta-calls, which we have adopted.

(1) The meta-call is compatible with unification in that if two terms unify then calling either will always invoke the same procedure (even if the terms come from different modules).

(2) All procedures have a term representation irrespective of their visibility.

(3) All procedures may be meta-called irrespective of their visibility.

As a consequence of these principles we cannot allow the clauses defining a global procedure to be in different modules. Consider the following counter-example:

```
module m1.          module m2.          module m3.
global p1,q.        global p2,q.        global p1,p2.

p1(q).              p2(q).              ?-p1(X), p2(Y),
q:-....             q:-....               X = Y, call(X),
                                          call(Y).

endmodule.          endmodule.          endmodule.
```

We have two predicates q with the same name and arity but in different modules. In each module we have a term representation of the predicate (principle 2), as an argument to a global procedure. Calling the global procedures with a free variable gives us the procedures' term representation, each of which we may meta call (principle 3) and invoke the corresponding version of q. If we declare both qs to be global as above, then their term representations must unify (since they are globals of the same name), but we should expect the calls of X and Y to call different clauses, in contradiction of principle 1. Thus we must declare one or both qs to be local, in which case they will not unify with each other, satisfying principle 1. Those systems that do not explicitly recognise the local terms either violate one of these principles or implicitly have local terms.

To Szeredi's principles we add a fourth:

(4) The BIPs such as univ/2 and functor/3 have a symmetric semantics when decomposing and recomposing terms.

This has the consequence that functors and predicates of the same name must all have the same visibility, regardless of their arity, within any module that references them. Consider the following counter-example where we have a global atom *foo* and a local functor *foo*/1.

```
module m.
global foo/0.
local foo/1.
?- foo(a) =..[Atom|Args], F =..[Atom|Args], foo(a) = F.
endmodule.
```

The first =.. unifies *Atom* with *foo*/0. Since *foo*/0 is global, having the same representation in all modules, it has no particular reference to the module *m*, and when used to construct *F* it produces a global *foo*/1 which will not unify with the local *foo*/1. Thus allowing the functors to have different visibilities means that decomposing a term and rebuilding it from the components is not guaranteed to produce a term compatible with the original!

Note that it is perfectly permissible to have the declaration *global foo* in modules other than *m*, and to have *local foo* inside *m*. Any reference to *foo* from within *m* will always refer to its local definition and not any external global definition. This facility is also useful when redefining system predicates within a module, as explained below.

We consider that the restriction in the use of names imposed by the atom-based visibility is justified by the consistent semantics we get for call, univ, functor and clause.

2.1 Default visibilities

All clauses enter the Prolog database either via the explicit use of assert or via an assert as part of consult. In order that this conversion of assert's argument from term to procedure should not side-effect its visibility we make the default visibility for term and procedure the same. For reasons of avoiding multiple definitions of procedures it is safe to assume a *default of local*. That is, all atoms, functors and predicates are local to a module unless a global declaration has been made for their atom. It should be noted that the string type has global visibility and provides us with a global atomic type which does not need declarations.

3. META-PREDICATES AFFECTING VISIBILITY: RESOLVE, EXPORT, IMPORT AND OF

Recall that we cannot have a global name that has predicate definitions in more than one module, since a call of their term representation would leave an ambiguity over which definition to use, as explained earlier. However, in large programming projects it is inevitable that groups of modules will be written, between which there are global name clashes. We consider that the modules should provide assistance in resolving such clashes. One way of doing this would be to have an import declaration that specifies the module from which the imported procedure is to come, but this has the drawback that it forces modules to make static references to each other. In prototyping and developmental environments there are often different versions of a procedure, and we want to bind these to the calls at consult time rather than at file creation (edit) time. An obvious solution is to rename all the clashing predicates, both where they are defined and where they are used, and we provide the *resolve* BIP for this purpose. Its syntax is of the form

*resolve Proc Mod*1:*ModList*1, *Mod*2:*ModList*2, ...

and it will rename the procedure *Proc*, defined in module *Mod*1 and called from the modules of *ModList*1, to some internal name and likewise rename *Proc* as it appears in the other *mod*:*Modlist* pairs to different internal names.

Typically a user will consult all the modules that he requires and then the system will inform him of any clashes, which can then be resolved with the resolve predicate. This is superior to the type of system which resolves clashes based on criteria which may be of no consequence to the user, e.g. Waterloo Prolog resolves clashes by the order in which the various modules were consulted.

It is possible to put *resolves* into files as directives along with *consult* directives and so package up a group of modules. We can also achieve a higher level of module structuring by having declarations that describe hierarchies or networks of modules and make the system automatically *resolve* clashes on the basis of these hierarchies.

3.1 Exporting and importing of global procedures

A definition of a procedure whose name is declared global, effectively *exports* it from a module, while a call

to a global procedure which does not have a definition effectively *imports* it. However, if we rely on implicit exporting and importing there is the danger that a procedure is exported unintentionally or that, by forgetting a definition, an import is made instead of an intra-modular call. We therefore have explicit *export* and *import* declarations and the system considers it an error if it finds a global procedure or call that has not been explicitly exported or imported. The export and import declarations make their arguments global if they are not already so, and hence the *global* declaration is only needed for a term or atom that needs to be unifiable across modules.

The export and import as described so far are very similar to those of MProlog, but further to this we allow *exports* that do not have any definitions. In Prolog the absence of a predicate can validly be used to indicate failure, likewise the non-existence of an optional predicate such as portray, and so by having export declarations without definitions we are able to extend this behaviour across modules. The assert predicate (Section 5.3) makes use of the export declarations when its argument is global, and thus global procedures can be created by the user at run time.

3.2 Uniform treatment of system names

A system predicate or atom is one that is exported from or declared global in the distinguished module *sys_mod*. These system names are automatically imported or declared global in the other modules that use them, thus relieving the user from a multitude of declarations. However, within a module a system predicate may be explicitly declared as local which overrides the generated declaration and allows a local version of the predicate to be used within the scope of the module.

Since the system predicates are declared in the same way as user predicates, albeit after automatic generation, we can use the resolve command to allow groups of modules to use alternative definitions of already defined BIPs. For example, suppose we have a new version of *listing* in the module *mx* which understands a different syntax and we want modules *m*1 and *m*2 to use this version then

```
?-resolve listing mx:[m1,m2], sys_mod:[m3,m4,m5].
```

would achieve this end with modules *m*3, *m*4 and *m*5 using the system's definition of listing.

3.3 Of for constructing local references

Having local names avoids accidental name clashes, but there may be occasions when we want to override the modularity and construct a reference to someone else's local name. An occasion where this arises is with predicates that have to be *distributed across* modules. Typical is *term_expansion*/2 which the user provides as a rewrite rule to be used during consult. For example, the definite clause grammar preprocessor can be invoked by having a *term_expansion* rule for '→', thus

```
term_expansion((A→B), (C:-D)):- dfg (A,B,C,D).
```

which succeeds by replacing $A \rightarrow B$ with a Prolog clause C:-D after calling *dfg*, the grammar preprocessor.

Each module may have its own *term_expansion* clauses. In a flat name space the system stores the term expansion clauses in an arbitrary order and relies on the fact that their argument values are usually disjoint. In Perlog we do not allow alternative definitions for a global predicate to be in more than one module, so we have to make the *term_expansion*s local predicates.

In order to reference predicates from other modules we have introduced a new BIP operator called *of* which takes a predicate and a module name as arguments, constructs a reference to the version of the predicate as it was declared in that module which it then calls. A definition of the *of* predicate is given as part of the model in Section 5. Thus a definition of *consult* that reads clauses from a file *F*, expands them if required, and then asserts them into a module *M* would be

```
consult(F,M):- see(F), read(X,M),
  (X==end_of_file;
    term_expansion(X,Y,M),
    assert(Y,M),
    fail),
  !, seen.
term_expansion(X,Y,M):- term_expansion(X,Y) of M,!.
term_expansion(X,X,_).
```

Read has a module argument so that it can create *X* with the correct visibility as declared, possibly by default, in *M* and *assert* has a module argument for reasons explained in the section on 'Semantics of Assert'.

Note that *of* can also be used for constructing references to a public predicate from a module where it is declared to be local, thus acting as an escape mechanism – see the example in Section 6.

4. MODULES AND ABSTRACT DATA TYPES (ADTs)

ADTs are used as a means of ensuring a modular approach to programming. The data types' representations are hidden within a module and only one set of predicates, called the ADT's methods, are supplied which can manipulate objects of the type – all access to the ADT has to go via these method predicates.

In our system we represent objects belonging to the same abstract data type as terms all with the same atom functor, which is declared local to the module containing the method definitions, for example:

```
module m.
  export newobj, method1, method2.
  private t.

  newobj(t(X,Y),X):- body(X,Y).
  method1(t(X,Y),A,B):- ...etc
  method2(t(X,Y),t(P,Q)):- ...etc
endmodule.
```

Since the method predicates have the local term representing the ADT in their head, they will not unify with any data type from another module, or any of the primitive data types. This is a simple and effective run-time type check. We have also used the declaration *private* which behaves like *local* but ensures that the meta-predicate *of* cannot be used to unify the term *t*/2 with terms from another module (see interpreter in Section 6).

An equivalent way of defining ADTs in Modules, but without using local terms, is described by Furukawa

et al.[3] We have extended it further to allow the method predicates to call out to the database systems language PS-Algol, passing the arguments of the local terms which are basic PS-Algol data types, and thus hidden from Prolog, which cannot destruct them or dereference them.[4]

4.1 Method declarations – dynamic local references based on ADTs

We have already mentioned that global predicates cannot be multiply defined, and if a clash occurs we must rename them. We have seen that in cases where the multiple definitions are actually alternatives rather than mistakes or redefinitions (e.g. *term_expansion*) we make them local instead of global and construct references via the *of* command.

In the example module just given we exported the methods as global procedures which must not clash with any others. However, we could instead declare the methods to be local, since in the case of ADTs the methods and terms belong to the same module, and so we can locate the methods based on the module of their argument and construct dynamic references to them using *of*. That is, since the argument to a method is an ADT, we can find the module that it came from and hence the appropriate method even when methods of the same name may be defined in more than one module.

Consider the *portray/1* predicate which is used to print objects in a specialised way. We can have several modules each implementing a different ADT and each with a different definition of *portray*. In each of the modules where portray is defined we may make the declaration *export_method portray* – this makes *portray* local, and in modules from where we want to run *portray* we have *import_method portray*.

The *import_method* declaration effectively plants the code

portray(*O*):-*module_of*(*O*,*M*), *portray*(*O*) *of M*.

Here *module_of* is a BIP that returns the module *label* from a local functor. Hence calls to *portray* use the definition based on the module of its local argument. Sufficient checks are needed to ensure that the argument to *portray* is bound and that the module in question did actually export *portray* as a method. Thus method procedures reduce the number of global procedures and hence the likelihood of a clash. Note that procedures that return the first instance of an ADT cannot be declared as methods since their ADT argument is unbound. In ESP there is a similar notion where method calls choose the actual method dynamically.[2]

5. MODULES AND PERSISTENCE

Persistence is the ability of objects to persist between runs of a program. Atkinson *et al.*[1] have developed the language PS-Algol, where persistence is potentially a property of all types in the language and where the transfers between main memory and disc are transparent. At the end of a session all the objects that are reachable from a given root are copied to disc with their in-store pointers replaced by file offsets; objects are retrieved from disc incrementally as and when they are referenced.

Perlog is a Prolog which interfaces with the PS-Algol language at one level and at a lower level to the Persistent Object System (POMS), which underlies PS-Algol. The higher-level interface enables us to experiment with different storage and access mechanisms, e.g. B-trees and hashing, and to manipulate the data types of PS-Algol, e.g. vectors and pictures.[7] The low-level interface allows the internal representation of the Prolog database to be migrated onto disc, preserving an isomorphic copy of the data structures for use in future sessions. This is the process of commitment. When a persistent Prolog database is restored we return to the state of the clause base as it was prior to the commit, but clause representations are only read in from disc as required, and no parsing is needed.

5.1 Modules as the unit of commitment

A common representation of Prolog programs is to have an atom dictionary containing atom and functor tokens which lead to clause representations made from more atom and functor tokens. With such a representation atom and functor comparison reduces to address comparisons and the clauses for a particular goal are easily found. All of the objects in the clause base are reachable from the dictionary and it is thus a convenient root for the commitment algorithm of POMS. Committing a clause base that has had changes made to it through calls to assert and retract will copy the changes to the persistent database.

To enable users to be selective in the changes that get committed we maintain a separate dictionary for each module and treat modules as the unit of commitment. Thus we can commit individual modules each to a separate object store. Import and export links are severed in the persistent module enabling them to be shared and recombined with different modules when they are restored. Thus we do not commit the effect of a *resolve*, which allows us to use a different choice of resolve in another session.

Commits are transactionally secure and POMS maintains a *One writer – many readers* concurrency protocol. The *commit* of Perlog is analogous to the *save* of other Prologs, e.g. CProlog,[9] in that a disc copy is made of an internal representation, but it is superior in that the modularisation of Prolog can be projected on to secondary store. Also because the object store is very general we can commit unit clauses that have bitmaps, vectors and even PS-Algol procedures as their arguments. The incremental fetching of disc objects is superior to the complete load of 'save'd images when we make relatively few accesses to a large clause base.

5.2 A Prolog model of our system

We present a simple Prolog model of the Perlog system in order to clarify its semantics. The model has been made as simple as possible and makes no attempt to be efficient. We first illustrate how a module is represented. Say we have the following module

```
module m.
  export p.
  global a.
  p(X):-foo(X).
  foo(a).
  foo(X):-q(X) of m1.
endmodule.
```

Then it is represented as a unit clause holding the name of the module, a list of all its clauses and lists of the various visibility declarations, excluding local which is the default, thus

```
module(m,
  [p(X) :- foo(X):m, foo(a):m :- true,
  foo(X):m :- of(q(X):m, m1:m)],
  exp([p]), imp([]), glo([p,a]), pri([])).
```

Notice that the local names *foo*, *q* and *m*1 have been suffixed with the module name and hence will not unify with *foo*s and *q*s from other modules. For the purposes of the model the suffix is bound to a name by the infix':' operator, but in the implementation this is actually achieved by making the dictionary entries have references back to their own dictionary. Since modules act as the unit of commitment we have gathered the clauses together under their module name, so that a commit of module *m* knows exactly which clauses are to be copied to disc. All clauses are associated with exactly one module.

To complete our model we need an interpreter, again written in ordinary Prolog. To keep things simple we assume that the code to be interpreted does not have any cuts or disjunctions and that unit clauses have been given bodies of *true*. The *do_goal* predicate is the meta-call predicate in the object language of this interpreter, and thus it serves to define the semantics of the *call* predicate of Perlog.

```
module(sys_mod, [true,...], exp([true,do_goak,of,...]), ...).
do_goal((L,G)) :- do_literal (L), do_goal(G).
do_goal(L) :- L\==(_,_), do_literal(L).
do_literal(X:M) :- do_abs_ref(X:M, M).
do_literal(X) :- X \== _:_,
  functor (X, Name, _), module(M, _, exp(L), _, _, _),
  member(Name, L), do_abs_ref(X, M).
do_abs_ref(do_goal(Goal), sys_mod) :-
  !, do_goal(Goal).
do_abs_ref(of(X,M), sys_mod) :-
  !, glo_rep(M, GM), $of(X, GM, XM), do_literal(XM).
do_abs_ref(X, sys_mod) :-
  host_pred(X), !, call (X).
do_abs_ref(X, M) :-
  module(M,L,_,_,_,_), member((X:-Goal),L), do_goal
  (Goal).
$of(X:_,M,X) :- X =.. [Name|Args], module(M,_,_,_,
  glo(G),_),
  member (Name,G).
$of(X:_,M,X:M) :- X =.. [Name|Args], module(M,
  _,_,_,glo(G),pri(P)),
  not(member(Name,G)), not(member(Name,P)).
$of(X,M,X) :- X=.. [Name|Args], module(M,_,_,_,glo
(G),_),
  member(Name,G).
$of(X,M,X:M) :- X =.. [Name|Args], module
  (M,_,_,_,glo(G),pri(P)),
  not(member(Name,G)), not(member(Name,P)).
member(X,[X|_]).   member(X,[_|L]) :- member(X,L).
glo_rep(X:_, X). glo_rep(X,X) :- X \== _:_.
```

The *do_goal* predicate calls *do_literal* on each conjunct from a conjunction of literals. *Do_literal* calls *do_abs_ref* either directly if its argument is a local term or else if it is a global term, after searching through export lists to find where it is defined. *Do_abs_ref* traps the term *of* and *do_goal* and calls these procedures directly. If its argument is a predicate which is implemented within the Prolog interpreter's host language, it calls *call*, otherwise it uses *member* to try unifying the literal *X* with the head of some clause whose body, *Goal*, is then passed to *do_goal*. Any instantiations made during the head unification are carried through the body *Goal*. The four clauses for $*of* handle all cases of local and global, input and output arguments.

Because the interpreter cannot interpret cuts, yet has cuts in itself, we have had to write it outside any module and trap calls to itself (1st *do_abs_ref clause*). The *of* predicate is best kept outside the module scheme so that the ability to construct local names is not generally available. We actually have a more complex interpreter that can handle cuts and disjunctions and hence we have been able to place it within the system module itself. A top-level interpreter sits outwith the module scheme and is used to read queries and invoke the *do_goal* of the module *sys_mod*.

5.3 Semantics of assert

In non-persistent module implementations global clauses can live in a global area, but because we have modules as a unit of commitment all clauses must live in some module. In our model of Perlog, an assert adds a clause to the clause list of a module. If the clause to be asserted is a local one then it is added to the module identified by the tag on the term, otherwise it is added to the module that has an export declaration for it. Asserting a clause has no visibility side-effects yet leaves the clause where it can be found by the interpreter.

Thus if *X* is a unit clause

```
assert(X), call(X)
```

will always invoke the procedure to which the asserted clause has just been added. This is the same as for flat Prologs.

Recall that the system is intended to support an exploratory style of programming and that the resolve predicate disambiguates name clashes from within Prolog rather than at the textual level. Consulting another module that redefines an existing global procedure will always succeed, but produces a state that needs to be 'resolve'd. In order to create such temporary inconsistent states we have introduced an assert predicate that takes a module name as a second argument which it uses to override the default behaviour of assert/1.

6. AN EXAMPLE – EXTENDING THE SYSTEM PREDICATES

The assert and retract predicates are responsible for maintaining Prolog's clause base and the call predicate initiates Prolog's inference mechanism. Kauffmann describes Multilog in which *add*, *remove* and *show* are user definable versions of these predicates and gives several examples which demonstrate the power of giving modules their own database and inference mechanism.[5] We choose the *add* predicate as an example and show that it can be easily defined in Perlog.

In *sys_mod* we have the following:

```
export add.
add(X,M):-add(X) of M, !.
add(X,M):-assert(X,M).
```

Add is a system predicate (exported from *sys_mod*) and hence will be potentially accessible from all other modules. The first clause says that we use the *add* which is local to *M* if it exists, and failing this to use the *assert* BIP.

Now consider a module *m* that wants to store student clauses in a special B-tree, with all other clauses being asserted normally.

```
module m.
  local add.
  student(Name, Address):- b_tree_lookup(student-
  (Name, Address)).

  add(X):- x==student(_,_), !, b_tree_add(X).
  add(X):-assert(X).

  add(X,M):- add(X,M) of sys_mod.
endmodule.
```

We declare *add* to be local so that it does not clash with the system's *add*. The first *add*/1 clause recognises a student term and stores it using some B-tree routine. All other types of term are 'assert'ed. Remember that local definitions hide global ones, so the *add*/2 clause is just there as an escape to the system's definition of *add*/2. There is one student clause which instantiates its argument via a B-tree lookup routine. Thus users can *add* clauses to the module *m*, via *add*(*X*,*m*) and be unaware that some of them are being stored in special way. *Remove* is defined in a way similar to that of *add*.

7. IMPLEMENTATION

Perlog runs on an amalgamation of CProlog and PS-Algol on a VAX-750 under UNIX. The persistence aspect of Perlog has already been implemented and is reported in Ref. 7. The modules have been prototyped in Prolog and are currently being rewritten in C. All of the original 'Prolog in Prolog' part of CProlog has worked correctly when run under the module scheme, and because of the devious nature of the bootstrap code we consider this a substantial test.

Using non-modular and non-persistent benchmarks the performance of Perlog is 20% slower than that of the original CProlog. This is attributed to generalisations that had to be made in order to bond the CProlog and PS-Algol interpreters together. Comparing the time for a complete commit of a new program with that of the analogous *save* of CProlog, it is an order of magnitude slower. However, for large databases the subsequent incremental retrieval is better than a complete restore.

8. CONCLUSION

We have described the Perlog system, which features both modules and persistence. The unit of visibility is the atom rather than the predicate and in this respect it is the same as MProlog and Micro-Prolog. It is particularly suited to an exploratory programming environment in that it allows modules to be committed separately, and then to be recombined in different configurations; it accepts multiply defined procedures and provides a BIP *resolve* for resolving the ambiguity; system predicates are subject to the same interface declarations as user predicates and they may be redefined within the scope of several modules. It has the BIP's private and method, which is sufficient to provide an ADT mechanism. It also has an assert BIP that takes a module argument and which gives control over where clauses get asserted so that the commitment of modules copies the correct clauses to disc.

Acknowledgements

David Moffat would like to thank the other members of the BSI Prolog Modules Sub-group, particularly David Bailey, Paul Chung, Kevin Cunnington, Jeff Dalton, Neil Davis, Tony Mansfield, Robert Rae and Roger Scowen, for many lively discussions which have given us a clearer understanding of the module visibility issues. It was also David Bailey who maintained that univ should behave symmetrically with respect to decomposition and recomposition.

REFERENCES

1. M. P. Atkinson, P. Bailey, W. P. Cockshott, K. J. Chisholm and R. Morrison, Progress with persistent programming. In *Databases – Role and Structure*, edited P. M. Stocker, P. M. D. Gray and M. P. Atkinson. Cambridge University Press (1984).
2. T. Chikayama, *ESP as a Preliminary Kernel Language of Fifth Generation Computers*. ICOT Technical Report TR-005 (1983).
3. K. Furakawa, R. Nakajima and A. Yonezawa, *Modularization and Abstraction in Logic Programming*. ICOT Technical Report TR-022 (1983).
4. P. M. D. Gray and D. S. Moffat, *A Prolog Extension to the Functional Data Model with Modular Commitment*. Internal Report, Department of Computer Science, University of Aberdeen, 1987.
5. H. Kauffmann and A. Grumbach, Representing knowledge within 'Worlds'. *Proceedings, First Expert Database Systems Conference*, edited L. Kerschberg. Charleston, S. Carolina (1986).
6. F. G. McCabe, *Micro-Prolog, Programmer's Reference Manual*. Logic Programming Associates Ltd (1981).
7. D. S. Moffat and P. M. D. Gray, Interfacing Prolog to a persistent data store, *Proceedings, Third International Conference on Logic Programming, London* (1986).
8. R. A. O'Keefe, Towards an algebra for constructing logic programmes. *Proceedings, Symposium on Logic Programming, Boston* (1985).
9. F. Pereira, D. Warren, D. Bowen, L. Byrd and L. Pereira, *C-Prolog User's Manual*, 1983. EdCADD, Department of Architecture, University of Edinburgh.
10. P. Szeredi, *Module Concepts for Prolog*. SZKI, 1251 Budapest P.O.B. 19 (1983).