Realizing Modularity in λProlog*

Gopalan Nadathur and Guanshan Tong
Department of Computer Science
The University of Chicago
Chicago, IL 60637
(773)702-3497, 2575
{*gopalan, guanshan*}*@cs.uchicago.edu*

September 1997

Abstract

The language λProlog incorporates a module notion that permits the space of names and procedure declarations to be decomposed into smaller units. Interactions between these units can take place through either an accumulation or an importation process. There are both static and dynamic effects to such interactions: the parsing of expressions may require names declared in another module and executable code may utilize procedures defined elsewhere. We describe a method for implementing this feature for modular programming that is based on the *separate* compilation of each module into an appropriate fragment of code. The dynamic semantics of module importation involves enhancing existing program contexts with the procedures defined in other modules. This effect is achieved through a run-time process for including the compiler generated code for such procedures. Our implementation method partitions the code space into distinct chunks corresponding to the module structure, with the management of the sub-parts being realized through a module table. Efficiency of execution requires the use of uniform structures, such as a common symbol table, for the code in all sub-components. To realize this requirement, we describe a suitable communication between the compilation and loading processes. The scheme presented here has been implemented and tested in conjunction with an abstract machine for λProlog.

Keywords: Prolog, λProlog, modularity, separate compilation, implementation.

1 Introduction

Logic programming based solely on the theory of Horn clauses lacks devices for structuring the space of names and procedure definitions. Although the absence of structuring facil-

*Work on this paper has been partially supported by NSF Grants CCR-9208465 and CCR-9596119. Comments on this work are welcome and may be sent to either author at the indicated addresses.

ities is not problematic in the development of small programs, it can become an issue in programming-in-the-large. This fact has stimulated several proposals of mechanisms that support the modular construction of programs. Bugliesi, Lamma and Mello [2] classify these proposals into two kinds: those that retain Horn clauses as the logical core but endow the framework with metalinguistic mechanisms for composing separately constructed program fragments, an approach exemplified by [1, 7, 13, 14], and those that extract composition and scoping mechanisms from an enrichment of the underlying logic, an approach exemplified by [3, 8, 10]. We consider in this paper the implementation of a modularity notion in the language λProlog that arises from following the latter approach [9].

Programming in a Prolog-like language to a large extent consists of identifying two kinds of collections: the names of constants, functions and predicates that can be used in constructing well-formed expressions, and procedure definitions that might be used in solving goals. The module concept that we consider is relevant to a structuring of programs with respect to both components. Modules in λProlog correspond, in a simplistic sense, to named collections of object identifiers and procedure definitions. The typical use of a module consists of making its content available in some fashion within another module or in the process of answering a query. This operation has both static and dynamic effects. The main impact of making the names of objects declared in a module visible is a static one: these names become relevant to parsing expressions in the new context. The effect with regard to procedure definitions is, on the other hand, largely dynamic. The procedure definitions in the new context might contain invocations to procedures defined in the "imported" module. To make sense of this notion, it is important to determine the exact manner in which procedure references are to be resolved in a situation where the available code is changing dynamically.

In implementing the form of modular programming described above, the main concern is that of handling the dynamic aspects. In particular, it is necessary to describe (a) run-time structures that will realize changing program contexts satisfactorily and (b) compilation and loading methods for extracting and setting up the information that is needed during execution. Such questions have been considered previously in the literature [4, 6, 11]. We expand on prior work in this paper in two significant ways. First, we provide a treatment of information hiding and a notion of module accumulation in addition to a previously considered notion of module importation. Second, in contrast to earlier discussions, we pay attention here to certain "low-level" details that are important to the actual execution model. We present our implementation ideas in the form of enhancements to an underlying Prolog engine such as the Warren Abstract Machine (WAM) [15] serving to realize the dynamic aspects of the module notion.[1] There are several interesting characteristics to the scheme we ultimately suggest, including the following:

1. Assuming some form of interface or signature definitions, it supports the separate compilation of modules.

2. It partitions the active code space into distinct chunks corresponding to the module

[1] In reality, λProlog has many other new features, each of which requires embellishments to the WAM [5, 11, 12]. We elide this aspect for simplicity in presentation.

structure and it manages this space through a global module table.

3. Despite the partitioning of code space, it realizes efficiency in execution by using uniform structures, such as a common symbol table, for all relevant code.

The remainder of this paper is structured as follows. The constructs for modular programming that are of interest to us derive in significant part from scoping devices already present in the core language. An appreciation of this relationship and of the method we have previously proposed for realizing the scoping notions [11] is useful in understanding the present implementation ideas. We provide this background in the next section. In Section 3, we present the important components of the modules language and we sketch the compilation and run-time processes needed to support it. A key ingredient of the scheme we present is a proper communication between the compiler and the loader. We explain some of the details of this aspect in Section 4. Section 5 concludes this paper.

2 Scoping Mechanisms in the Core Language

The higher-order theory of hereditary Harrop formulas underlies the core language of λProlog. From a practical perspective, this logic differs from that of Horn clauses in that it incorporates a typing discipline, supports higher-order programming and provides for scoping mechanisms. The scoping constructs that are of chief interest here result from extending goals in the Horn clause setting with two new logical primitives, those of universal quantification and implication. The first allows an expression of the form $\forall x G$ to be considered as a goal, assuming that G is itself a goal. Operationally, solving such a goal involves introducing a *new* constant c, replacing the variable x everywhere in G with this constant and then solving the resulting goal. Viewed differently, the universal quantifier corresponds to giving a name a scope: in the goal in question, the "name" x has G as its lexical scope and the duration of the attempt to solve G as its dynamic scope. The second new primitive permits an expression of the form $D \supset G$ to be a goal, where D is the conjunction of program clauses, and G is, once again, a goal. The operational interpretation of such a goal is that the program is to be enhanced with the "partial procedure definitions" in D while attempting to solve G. Implication is, thus, a primitive for giving procedure definitions a scope.

The programming character of the new primitives is illustrated by their use in local definitions, a typical application of scoping notions in programming-in-the-small. The following clause defines the *reverse* predicate on lists:

$$
\begin{array}{l}
reverse(L1, L2) \;\text{:-}\\
\quad (\forall rev\_aux\, ((\, rev\_aux([\,], L2) \wedge\\
\qquad\qquad\qquad (\forall X\, \forall L1\, \forall L3\, (rev\_aux([X|L1], L3) \;\text{:-}\; rev\_aux(L1, [X|L3]))))\\
\qquad\qquad\qquad\quad \supset rev\_aux(L1, [\,]))).
\end{array}
$$

The body of this clause contains a universal goal whose body is itself an implication goal. Notice that universal quantification over a program clause that is usually left implicit must be made explicit when the clause appears in the antecedent of an implication goal. Suppose now that the goal $reverse([1,2],L)$ is invoked relative to the given procedure definition. This would lead to an attempt to solve the goal

$$\begin{array}{l} \forall rev\_aux\,((\,rev\_aux([\,],L) \wedge \\ \qquad\qquad (\forall X\,\forall L1\,\forall L3\,(rev\_aux([X|L1],L3) \;\text{:-}\; rev\_aux(L1,[X|L3])))) \\ \qquad\qquad\qquad \supset rev\_aux([1,2],[\,])). \end{array}$$

The semantics of the universal quantifier dictates treating rev_aux as a new name, distinct from everything else in the embedding context. Processing the body of the universal goal leads to the invocation of the goal $rev\_aux([1,2],[\,])$ after the clauses

$$\begin{array}{l} rev\_aux([\,],L), \quad \text{and} \\ (\forall X\,\forall L1\,\forall L3\,(rev\_aux([X|L1],L3) \;\text{:-}\; rev\_aux(L1,[X|L3]))) \end{array}$$

have been added to the program. Universal quantifiers have been retained in the second clause above to distinguish the variables they bind from the variable L in the first clause that is really a "global" bound to a value occurring in the original query. Notice that the computation described up to this point has introduced a definition for the procedure rev_aux and that this must be used in solving a new goal. Carrying this process to fruition will eventually result in the query variable L being bound to $[2,1]$.

New devices are needed for implementing the scoping primitives described above. It appears at the outset that rather simple machinery might suffice for universal goals: we merely instantiate universal quantifiers with newly generated constants. However, this scheme is not quite sufficient because universal and existential quantifiers can appear in arbitrary order in goals. For example, suppose that our program consists solely of the clause $\forall x\, p(x,x)$ and that we are interested in solving the goal $\forall y\, p(X,y)$. The variable X has existential strength in the given goal in that it might be instantiated as desired in the course of search. Notice, however, that the interpretation of universal goals prohibits the instantiation for X from depending on any value chosen for the variable y. Now, the suggested treatment would reduce the given goal to $p(X,c)$, assuming that c is a new constant. At this point, the usual notion of unification would result in success by incorrectly instantiating X to c.

The solution to this problem is to modify unification to respect quantifier order. There is a simple way to achieve this that is based on tagging variables and constants with (small) positive integers. We sketch this scheme here, referring the reader to [11] for details. The key step is to think of the collection of all terms as being arranged in an increasing hierarchy of universes. The level 1 universe consists of all the constant symbols that appear in the program clauses and the original goal. These symbols are tagged by the number 1 to indicate their position in the hierarchy. Each time a universal quantifier is processed, a new constant is introduced, giving rise to the next universe in the hierarchy. This requirement is accounted for by increasing a running "universe index" by 1 and generating a new constant tagged with this index. The collection of constants at the new level consists of all those constants tagged with a number less than or equal to that level. When an essential existential quantifier is encountered, it is instantiated by a logic variable. This variable is tagged with the current value of the universe index to indicate that it may be instantiated only by terms in the universe at that level. The actual use of the tags occurs when an attempt is made to bind a variable X with a tag i to a term t. This binding is permitted only if t does not contain any constants with a tag value greater than i.

The main issue in implementing implication goals is to provide an efficient realization of changing program contexts. The essence of the scheme described in [11] is in viewing a program as a composite of compiled code and a layered access function to this code, with each implication goal causing a new layer to be added to an existing access function. Thus, consider goal of the form $(C_1 \wedge \ldots \wedge C_n)$ `=>` G where, for $1 \leq i \leq n$, C_i is a closed program clause.[2] The C_is are to be added to the front of the program while attempting to solve G. These clauses can be treated as an independent program fragment and compiled as such. Let us suppose that the clauses define the predicates $p_1, \ldots, p_r$. The compilation process then results in a segment of code with r entry points, each indexed with the name of a predicate. In our context, we require compilation to also produce a procedure that we call $find\_code$ that performs the following function: given a predicate name, it returns the appropriate entry point in the code segment if the name is one of $p_1, \ldots, p_r$ and fails otherwise. The execution of the implication goal results in a new access function that behaves as follows. Given a predicate name, $find\_code$ is invoked with it. If this function succeeds, then the code location that it produces is the desired result. Otherwise the code location is determined by using the access function in existence earlier.

The process of enhancing a context described above is incomplete in one respect: the new clauses provided for $p_1, \ldots, p_r$ may in fact be adding to earlier existing definitions for these predicates. To deal with this situation, the compilation process produces code for each of these predicates that does not fail eventually, but instead looks for code for the relevant predicate using the access function existing earlier. To realize this function, we associate a vector of size r with the implication goal, the ith entry in this vector corresponding to the predicate p_i. The compilation of the body of the implication goal then creates a procedure called $link\_code$ that serves to fill in this vector when the implication goal is executed. This procedure essentially uses the name of each of the predicates and the earlier existing access function to compute an entry point to available code or, in case the predicate is previously undefined, to return the address of a failing procedure.

In the framework of a WAM-like implementation, the layers in the access function described above are realized by using a data structure called an *implication point record* that is allocated on the local stack. The components of such a record are the following:

1. the address of the $find\_code$ procedure corresponding to the antecedent of the implication goal,

2. a positive integer r indicating the number of predicates defined by the program clauses in the antecedent,

[2] In reality, the C_is may contain free global variables as in the case of the $reverse$ program discussed earlier. This requires treating a program clause as a combination of compiled code and a binding environment, but leaves other details unchanged.

3. a pointer to an enclosing implication point record, and thereby to the previous layer in the access function, and

4. a vector of size r that indicates the next clause to try for each of the predicates defined in the antecedent of the implication goal.

The program context existing at a particular stage is indicated by a pointer to a relevant implication point record which is contained in a register called I. Now, a goal such as $(C_1, \ldots, C_n)$ `=>` G is compiled into code of the form

push_impl_point t
  { Compiled code for G }
pop_impl_point

In this code, t is the address of a statically created table for the antecedent of the goal that indicates the address of its *find_code* and *link_code* procedures and the number of predicates defined. The *push_impl_point* instruction causes a new implication point record to be allocated. The first three components of this record are determined immediately from the table parameter and the I register. The last component is determined by running *link_code* relative to the access function provided by the I register. The final action of the instruction is to set the I register to point to the newly created implication point record. The complementary *pop_impl_point* instruction restores the old program context by setting the I register to the address of the enclosing implication point record.

An important operation is the resurrection of an old program context upon backtracking. In the scheme described, the program context at each point is encapsulated in the contents of the I register. Saving the contents of this register in a WAM-like choice point record and retaining implication point records embedded under choice points therefore suffice for achieving the necessary context switching.

3 The Structure of Modules and their Interactions

At the lowest level, the module feature allows a name to be associated with collections of names and program clauses. An example of the use of this construct is provided by the following code that attaches the name *lists* with the predicate names *append* and *member* and the procedures defining them:[3]

$module \quad lists.$
$append([\,], L, L).$
$append([H|L1], L2, [H|L3])$:- $append(L1, L2, L3).$
$member(H, [H|L]).$
$member(X, [H|L])$:- $member(X, L).$

[3] In reality, λProlog is a typed language and so named objects are identified explicitly through type declarations in modules. However, we suppress types in the present discussion.

The module declaration above can be thought of as the declaration of a list "data type". This data type can be made available in relevant contexts by using the name *lists* in specific ways. One possibility is to use the names of modules in a new kind of goal called a module implication. These are expressions of the form M ==> G, where M is a module name. The static effect of such a goal is that the names associated with M become available in interpreting the goal G. Of greater interest from an implementation perspective is the effect with regard to procedure definitions. This aspect is explained by a translation into the core language. We first interpret a module as the conjunction of the program clauses appearing in it. For example, the *lists* module corresponds to the conjunction of the clauses for *append* and *member*. Now, if module M corresponds to the formula D, then the query M ==> G is thought of as the goal $D \supset G$. The run-time treatment of the goal M ==> G thus calls for solving the goal G after adding the predicate definitions in the module M to the existing program.

The modules language incorporates the possibility of hiding data structures. In particular, a declaration of the form

$$local \quad \langle constant\text{-}name \rangle, \ldots, \langle constant\text{-}name \rangle.$$

can be placed within a module to achieve this effect: the names of the constants listed then become unavailable outside the module. The static effect of the local construct is obvious. From a dynamic perspective, another issue arises: the constants defined to be local should not become visible outside through computed answers. This effect can be achieved by interpreting local constants as variables quantified existentially over the conjunction of program clauses in the module. For example, consider the following module:

$$
\begin{array}{l}
module \quad\quad store. \\
local \quad emp, stk. \\
initialize(emp). \\
enter(X, S, stk(X, S)). \\
remove(X, stk(X, S), S).
\end{array}
$$

This module implements a *store* data type with initializing, adding and removing operations. At a level of detail, the store is implemented as a stack. However, the intention of the local declarations is to hide the actual representation of the store. Now, from the perspective of dynamic effects, the module can be interpreted as the formula

$$
\begin{array}{l}
\exists Emp \exists Stk(\\
\quad\quad initialize(Emp) \wedge \\
\quad\quad \forall X \forall S \; enter(X, S, Stk(X, S)) \wedge \\
\quad\quad \forall X \forall S \; remove(X, Stk(X, S), S)).
\end{array}
$$

If we call this formula $EStore$, then solving a goal of the form ($store$ ==> $G(X)$) amounts to solving the goal ($EStore \supset G(X)$). Operationally, the existential quantifiers at the head of $EStore$ can be translated into universal quantifiers over the implication to produce a legitimate goal in our core language. Notice, however, that these quantifiers are scoped *within* the (implicit) quantification over X and so the constants supplied for Emp and Stk may not appear in instantiations of X.

A module may interact with others in λProlog by *importing* the definitions appearing in them. A declaration of the following form is utilized for this purpose:

$$import \qquad M1, \ldots, Mk.$$

In such a declaration, $M1, \ldots, Mk$ must be names of other modules, referred to as the *imported* modules. This declaration has, once again, a static and a dynamic effect on the module in which it is placed, *i.e.*, on the *importing* module. The intended dynamic effect is to make the procedure definitions in the imported modules available for solving the goals in the *bodies* of program clauses that appear in the importing module. The exact effect can be clarified by using module implication [8]. Suppose that the clause P :- G appears in a module that imports the modules $M1, \ldots, Mk$. The dynamic semantics involves interpreting this clause as the following one instead:

$$P \texttt{ :- } (M1 \texttt{ ==> } \cdots (Mk \texttt{ ==> } G)).$$

Using this clause requires solving the goal $(M1 \texttt{ ==> } \cdots (Mk \texttt{ ==> } G))$ that ultimately causes the program to be enhanced with the clauses in $M1, \ldots, Mk$ before solving G.

The translation of *import* and *local* declarations into the core language indicates a first approach to their implementation. Suppose that a module M imports the modules $M1$ and $M2$. At a conceptual level, the (separate) compilation of $M1$ and $M2$ should associate with these modules the following components:

1. The number of local constants and, if this is nonzero, the names of these constants.

2. The number and names of predicates defined in this module that could be extending previously existing definitions.

3. Analogous to the antecedents of implication goals, the $find\_code$ and $link\_code$ routines for the module.

Let the addresses of these "tables" of information for $M1$ and $M2$ be $t1$ and $t2$, respectively. Then the compilation of the body of the clause P :- G in M would produce the following code:

push_import_point $t1$
push_import_point $t2$
    {Compiled code for invoking goal G}
pop_import_point 2

Using its table argument, the *push_import_point* instruction first determines if there are local constants for the module in question. If so, it increments the running universe index and tags the local constants with the new value. It then sets up an *import point record*, a structure similar to an implication point record, to register the addition of the code in the module. The *pop_import_point* instruction complements this operation by removing a number of import point records indicated by its argument and decreasing the universe index as needed.

The scheme described above is naive in the following respect: if clauses from an importing module are used more than once, it will repeat the addition of code for the importing modules and this is shown to be redundant in [4].[4] To avoid this, we augment import point records with a new cell, called a *backchained* cell, that indicates whether or not a clause from the module has previously been used. Further, we add two new instructions, *add_imports* and *remove_imports* that use the value stored in this cell to condition the pushing and popping of import point records.

module $m1$.	*module* $m2$.	*module* $m3$.
import $m2$.	*import* $m3$.	
	local r.	
q :- p.	p :- w.	w :- s.
s :- r.		r.

Figure 1: Interactions between Local and Import

There is an interaction between local and import declarations that needs to be considered carefully in implementing the scheme described. The particular problem arises from the fact that a module may use a local declaration to hide a (predicate) name defined in a module imported by it. The collection of module definitions shown in Figure 1 serves to illustrate this point. Suppose that the goal q is invoked in a program context in which the code from module $m1$ is available. This goal would ultimately fail. This situation is to be contrasted with one in which the predicate r is *not* defined to be local in module $m2$; in this case, the same goal would, in fact, succeed. Thus, the interpretation of a constant in module $m3$ depends on the context into which it is imported, raising the spectre of having to predicate compilation on usage. Fortunately, there is a solution to this problem: we compile each module separately, but in a way that allows the loading process to relativize it to the context of use. In the example being considered, the module $m3$ will be compiled as though the constant r that appears in it is a global. However, loading this module through $m2$ will cause this global to be "renamed" to a local. The actual implementation of this idea is discussed further in the next section.

There is one additional method for composing modules: several independently defined components can be accumulated into one larger unit. This effect is achieved by using a declaration of the form

accumulate $M1, \ldots, Mk$.

[4]This demonstration is a little subtle because of the presence of local constants.

module m1.	*module m2.*
	accumulate m1.
local a.	
$q(a)$.	$p(b)$:- $q(a)$.
	$r(b)$:- $q(X)$.

Figure 2: Interactions between Local and Accumulate

where $M1, \ldots, Mk$ are module names. Such a declaration can, for the most part, be treated by inlining the contents of the accumulated modules prior to the compilation of the module in question. However, there are a few subtle points relating to the scopes of *local* and *import* declarations that must be borne in mind. To understand this matter, we first consider the set of module definitions shown in Figure 2. The semantics of *accumulate* requires us to treat module $m2$ as a composite of the clauses for p and r that appear in it and the code in $m1$. However, the quantificational force of *local* requires us to distinguish between the constants of name a that appear in $m1$ and $m2$. Thus the goal $p(b)$ posed in the context of module $m2$ should *not* succeed. We note, nevertheless, that the scope of the local declaration in $m1$ should be raised to that of all of $m2$ by the composition process. In particular, the goal $r(b)$ should succeed relative to the module $m2$.

module m1.	*module m2.*	*module m3.*	*module m4.*
	import m1.		*accumulate m2, m3.*
w.	p :- w.	t :- w.	q :- p.
			r :- t.
			s :- w.

Figure 3: Interactions between Import and Accumulate

A similar point concerning the scope of a *import* is brought out by the modules defined in Figure 3. The accumulation of $m2$ and $m3$ into $m4$ results in the clauses in these modules being added to those already present in $m4$. However, this accumulation is to be done *after* the "de-sugaring" of syntax corresponding to *import* declarations. Thus, different sets of modules are to be imported for the purpose of solving the bodies of clauses arising from $m2$, $m3$ and $m4$. As a specific example, the goal q should succeed relative to the module $m4$ whereas both r and s should fail. The reason for this is that the code in module $m1$ should be added to the program context only when the clause p :- w in module $m4$ that originates from module $m2$ is used, and the (sub)goal w fails without this addition.

The run-time machinery that we have already presented suffices for handling most of the scoping aspects discussed above and the main requirement is to account for them satisfactorily in the compilation process. We explain how this is done in the next section. However, some modification to the implementation scheme is needed for importing different sets of modules based on the clause being used. Fortunately, a simple generalization of existing devices suffices for this: instead of a single backchained cell, we associate a *vector* of such cells with each import point record. The instructions *add_imports* and *remove_imports* acquire an index into this vector and use the cell determined by it to decide whether or not to execute a sequence of *push_import_point* instructions or a *pop_import_point* instruction, respectively.

4 Compilation and Loading of Modules

In this section, we discuss how modules are compiled and loaded. Before going into details, we give a general description of what should be produced in loading a module and how such information is used subsequently. A user asks for a module to be loaded by typing in a directive of the form:

$\#load \quad \langle module\text{-}name \rangle.$

The outcome of carrying out this directive is an association between the named module and the following items: (1) the global constants of this module and (2) a table created by the loader for adding the code from this module to the program context when the module is assumed. This association is realized through a global *module table*. Both items get used after the user issues a directive of the form:

$\#query \; \langle module\text{-}name \rangle.$

After such a directive, the system will be ready for user queries. The first item associated with the module will be used in parsing these queries and, upon successful parsing, the module relevant code will be used in answering them. Thus, this directive corresponds to relativizing queries through a module implication.

4.1 Separate Compilation of Modules

There are interactions between modules that require the sharing of information such as the names (and types) of constants at compile time. These interactions are typically handled by associating interface or signature declarations with modules and our implementation of λProlog incorporates such a notion. Armed with such information, our implementation scheme supports the independent generation of the code needed at run-time for each module.

The interactions between *local*, *import* and *accumulate* declarations discussed in Section 3 require the preprocessing of the code in modules prior to compilation. Figure 4 contains a schematic presentation of a set of module definitions that serve to illustrate all the significant aspects of the preprocessing phase. Based on these declarations, module $m5$ will be transformed into the form shown in Figure 5. Some explanations are in order with Figure 5. Module $m5$ eventually has four local constants: q, coming from the explicit *local* declaration

$module\ m1.$ $\ldots$

$module\ m2$ $\ldots$

$module\ m3.$
$import\ m1.$
$local\ r.$
$[clauses\ in\ m3]$

$module\ m4.$
$import\ m2.$
$local\ r, w.$
$[clauses\ in\ m4]$

$module\ m5.$
$accumulate\ m3, m4.$
$local\ q.$
$[clauses\ in\ m5]$

Figure 4: A module collection illustrating preprocessing

$module\ m5.$
$local\ q, r[1], w, r[2].$

$\{import\ m2\ [q \rightarrow$ *first local,* $r \rightarrow$ *second local,* $w \rightarrow$ *third local*$]$
$[clauses\ from\ m4]$ *with constant references suitably resolved*$\}$

$\{import\ m1\ [q \rightarrow$ *first local,* $r \rightarrow$ *fourth local*$]$
$[clauses\ from\ m3]$ *with constant references suitably resolved*$\}$

$[clauses\ from\ m5]$ *with constant references suitably resolved*

Figure 5: Preprocessed form of the definition of module $m5$

in $m5$, r and w coming from the declaration in $m4$ and r coming from the declaration in $m3$. This is reflected in the local declaration in the preprocessed form, with $r[1]$ and $r[2]$ indicating the "versions" of r arising from distinct sources. The clauses in the composite form of $m5$ can be partitioned into three sets depending on the import statements that are scoped over them. This structure is made explicit in Figure 5. When the imported modules are loaded into core, some of their global constants will have to be "captured" by the local declarations in $m5$. The loading process will realize this effect, as we explain shortly. However, the compiler must supply the information needed for constructing the renaming function for each such module. The fragments of this function determined by the declarations in $m5$ are indicated within square brackets adjacent to each *import* declaration. A final aspect relates to avoiding inappropriate capture of globals in clauses arising from different sources by the (expanded set of) locals. For example, the clauses in the original presentation of $m5$ may use the names r and w. In this case, some renaming must be done to distinguish these constants from those appearing in the *local* declaration in the composite form. This form of renaming is indicated schematically adjacent to each set of clauses in Figure 5 and would typically be built into the compilation phase.

The compilation of the preprocessed form of a module eventually produces the following components of information:

1. A list of the global constants that are defined and/or used in this module.
2. A list of the local constants that are identified in this module.
3. A count of the segments of clauses that have different importations scoped over them.
4. A list of imported modules paired with partial renaming functions for their global constants. As indicated, these renaming functions are realized as a list of pairs of a name and an offset into the local constant list of the present module.
5. The number and names of predicates (represented as offsets into the list of global constants) in the module that could be extending previously existing definitions.
6. The $find\_code$ function for the module. (The $link\_code$ function is completely determined by the previous component.)
7. The byte-code for the clauses in the module. Constant references in this code are provided as offsets into the global and local constant lists.

4.2 Loading Modules

The process of loading a module at the top-level will also involve loading imported modules, possibly triggering further nested loads. There is a difference in final effect between a top-level load and nested loads since nested loads do not produce a module name that is visible at the top-level, i.e., one having an entry in the global module table. As discussed in Section 4.1, constants in a module loaded in a nested fashion may need to be renamed in a way that depends on the context of importation. The loading process realizes this requirement by, in effect, loading specialized copies of modules. The relevant copy that is to be created at any point is determined by an incoming renaming function and the addition to this function that is made by the module from where the one being loaded is imported. The renaming function starts out by being empty for top-level loads.

The idea of loading a specialized copy of an imported module finds a correspondence in many other languages. For example, in C++, templates allow generic functions to be defined once for a family of types and the implementation will generate versions of such generic functions for each argument type supplied to the template.

4.2.1 Translation into absolute addresses

The structure of the loading process leads to a distinct code space for each module. References to instruction addresses within the code generated for a module is, however, relativized to a start address. The loader translates these into absolute addresses. This translation process affects the (byte)code that is loaded for clauses, the $find\_code$ function generated for the module and, ultimately, the vector stored in import and implication point records for determining the next clause to try for each predicate whose previously existing definition is being extended.

4.2.2 Resolution of global and local constants

A common symbol table is used for all the modules that are loaded during an interactive session. A benefit of doing this is a significant simplification in the run-time manipulation of constants. For example, equality testing of constants is reduced to a comparison of symbol table indices.

An important component of realizing this particular treatment of constants is their translation to symbol table indices. The process for local constants is simple: when a module is loaded either at the top-level or due to importation, each of its local constants is inserted as a new entry into the symbol table. The symbol table index that is determined by this process is used in loading the code for the clauses in the module and also in the renaming function that affects the loading of modules that are imported.

When a top-level module is being loaded, each of its global constants is mapped onto a fresh cell in the symbol table. Before an imported module is loaded, the existing renaming function is extended by utilizing the information generated by the compiler for this purpose. Note that the renaming function can be managed in a stack fashion, with the addition made to it in the course of loading an imported module being popped off after the loading is completed. Note also that the renaming function can be represented as a mapping from constant names to the symbol table indices determined for these (local) constants. Now, when a non top-level module is loaded, for each of its global constants, we search the incoming renaming function to see if it can be resolved with a local constant of one of the "ancestor" modules. If this is the case, the function yields the symbol table index to be used for the constant. If not, the incoming collection of global constants is searched to determine if an index has already been assigned to the constant. If so, we, once again, have the index that is to be used. If neither of these processes yields an index, then we create a new entry in the symbol table for the constant and use the index so generated for the constant.

For efficiency in parsing user queries and in the loading process, the collection of global constants is organized into a binary tree. A pointer to the root of this tree is retained in the global module table entry for each top-level module.

4.2.3 Creation of tables needed for adding module code to the program context

When a module is loaded, a table of information of the kind needed for it in conjunction with a *push_import_point* instruction is generated and retained in the corresponding code space. The way in which this table is constructed should be obvious. The loading process eventually returns a pointer to this table. For a top-level module, this pointer is saved in its entry in the global module table, to be utilized in when a *query* directive is encountered. For imported modules, this pointer is used to translate the table argument of *push_import_point* instructions in the importing module into an absolute address.

5 Conclusion

We have presented a module notion in the logic programming language λProlog, and described an approach to its implementation. We have shown that our implementation ideas support the notion of separate compilation even in a situation where the context of use determines the run-time interpretation of modules. We have also discussed some of the details of the compilation and loading processes used in our approach that are intended to facilitate efficiency of execution.

The ideas described in this paper have been implemented in conjunction with an abstract machine for λProlog. This abstract machine utilizes the basic structure of the WAM in the treatment of nondeterminism and unification. However, it also represents a considerable enhancement to the WAM, with mechanisms being added to treat features such as typing [5], scoping over names and procedure definitions [11] and lambda terms as data structures with higher-order unification being used as the corresponding destructuring operation [12]. A satisfactory realization of all these features requires many low-level decisions, such as those pertaining to term representation, to be different from those adopted in typical WAM emulators. We have, for this reason, not been able to use such emulators in our implementation and have constructed an entirely new emulator for our abstract machine in C. We have tested key features of our implementation, including ones corresponding to the modularity features described here, by hand-compiling and running a carefully chosen set of λProlog programs. We are currently implementing a compiler for this language. We plan to use the system that will result at the completion of this work to obtain an experimental validation of the efficacy of our ideas as well as to determine the actual benefits of the modularity notion in programming-in-the-large.

References

[1] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition operators for logic theories. In J.W. Lloyd, editor, *Computational Logic, Symposium Proceedings*, pages 117–134. Springer-Verlag, 1992.

[2] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19,20:443–502, May/July 1994.

[3] L. Giordano, A. Martelli, and G.F. Rossi. Extending Horn clause logic with modules constructs. *Theoretical Computer Science*, 95:43–74, 1992.

[4] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing a notion of modules in the logic programming language λProlog. In Evelina Lamma and Paola Mello, editors, *Extensions of Logic Programming: Proceedings of the Third International Workshop, Bologna, Italy*, pages 359–393. Springer-Verlags, 1993. Vol 660 of Lecture Notes in Artificial Intelligence.

[5] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.

[6] Evelina Lamma, Paola Mello, and Antonio Natali. An extended Warren abstract machine for the execution of structured logic programs. *Journal of Logic Programming*, 14:187–222, 1992.

[7] P. Mancarella and D. Pedreschi. An algebra of logic programs. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 1006–1023, Seattle, Washington, August 1988. MIT Press.

[8] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[9] Dale Miller. A proposal for modules in λProlog. In Roy Dyckhoff, editor, *Proceedings of the 1993 Workshop on Extensions to Logic Programming*, pages 206–221. Springer-Verlag, 1994. Volume 798 of Lecture Notes in Computer Science.

[10] Luís Monteiro and António Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 284–299, Lisbon, Portugal, June 1989. MIT Press.

[11] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, November 1995.

[12] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.

[13] Richard O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *1985 Symposium on Logic Programming*, pages 152–160, Boston, Massachusetts, 1985. IEEE Computer Society Press.

[14] D.T. Sannella and L.A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147–178, January 1992.

[15] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.