Modules for Prolog Revisited

Rémy Haemmerlé and François Fages

Projet Contraintes – INRIA Rocquencourt – France
FirstName.LastName@inria.fr

Abstract. Module systems are an essential feature of programming languages as they facilitate the re-use of existing code and the development of general purpose libraries. Unfortunately, there has been no consensual module system for Prolog, hence no strong development of libraries, in sharp contrast to what exists in Java for instance. One difficulty comes from the *call* predicate which interferes with the protection of the code, an essential task of a module system. By distinguishing the called module code protection from the calling module code protection, we review the existing syntactic module systems for Prolog. We show that no module system ensures both forms of code protection, with the noticeable exceptions of Ciao-Prolog and XSB. We then present a formal module system for logic programs with calls and closures, define its operational semantics and formally prove the code protection property. Interestingly, we also provide an equivalent logical semantics of modular logic programs without calls nor closures, which shows how they can be translated into constraint logic programs over a simple module constraint system.

1 Introduction

Module systems are an essential feature of programming languages as they facilitate the re-use of existing code and the development of general purpose libraries. Unfortunately, there has been no consensual module system for Prolog, hence no strong development of libraries, in sharp contrast to what exists in Java for instance.

One difficulty in Prolog comes from the *call* predicate which interferes with the protection of the code, an essential task of a module system. There has been therefore several proposals of module systems realizing different trade-offs between code protection and the preservation of meta-programming facilities.

In order to enforce the proper segmentation of the code, and to guarantee the semantics of the predicates defined in a library, a module system has however to strictly prevent any predicate execution not allowed by the programmer. This means that it should be possible to restrict the access to the code of a module (by predicate calls, dynamic calls, dynamic asserts or retracts, syntax modifications, global variable assignments, etc.) from extra-modular code. This property is called *code protection*.

The relationship between the calling module and the called module is however asymmetric. The *called module code protection* ensures that only the visible

predicates of a module can be called from outside. The *calling module code protection* should ensure that the called module does not call any predicate of the calling module, as they are not visible. The following example illustrates however the need to provide an exception to this rule with a mechanism for executing a predicate in the calling environment, which we will call a *closure*.

Example 1. The list iterator predicate `forall/2` defined below in ISO-Prolog [13], checks that every element of a list, passed in the first argument, satisfies a given property, passed as a unary predicate in the second argument:

```
forall([], _).
forall([H|T], P):- G=..[P,H], call(G), forall(T, P).
```

Such a predicate `forall` cannot be defined in a library (for lists as it should) without violating the calling module code protection, as the intended meaning of the predicate is indeed to call the predicate passed as argument in the calling module environment.

Most module systems for Prolog solve the difficulty either by abandoning any form of code protection, or by introducing *ad-hoc* mechanisms to escape from the code protection rules. Our proposal here is to keep a strict code protection discipline but distinguish closures from dynamic calls, closures being executed in the environment where they are created. From a functional perspective, a closure here is basically a lambda expression with only one parameter, i.e., that `closure(X,G,C)`) is somehow equivalent to $C = (\lambda X.G)$ and `apply(C,X)` to $C.X$. This makes it possible to define a module for lists which exports a `forall` predicate taking a closure from outside as argument.

Example 2.
```
:- module(lists, [forall/2, ...]).
forall([], C).
forall([X|T], C) :- apply(C, [X]), forall(T, C).
```
That definition of `forall` using closures instead of dynamic calls can be used from any module importing the list module, by passing to it a closure constructed from a unary predicate like `var/1` for instance:

```
:- module(foo, ...).
:- use_module(lists).
all_variables(L) :- closure([X],var(X),C), forall(L,C).
```

In this paper, we first review the main module systems for Prolog in the light of the two forms of module code protection. We show that no module system ensures both forms of code protection, with the noticeable exceptions of Ciao-Prolog and XSB.

Then we give a formal presentation of a safe module system with calls and closures. We precisely define its operational semantics and show the full code protection property.

We also provide an equivalent logical semantics for modular logic programs without calls nor closures. That semantics, obtained by translating modular logic

programs into constraint logic programs over a simple constraint module system, shows how the module system can be compiled into a constraint logic program.

We then conclude on the relevance of these results to an on-going implementation of a fully bootstrapped constraint programming language, from which this work originated.

Related work

Modularity in the context of Logic Programming has been considerably studied, and there has been some standardization attempts for ISO-Prolog [14]. Different approaches can be distinguished however.

The *syntactic approach* mainly deals with the alphabet of symbols, as a mean to partitionate large programs, safely re-use existing code and develop general purpose libraries. This approach is often chosen for its simplicity and compatibility with existing code. For instance, a constraint solver like OEFAI CLP(q,r) [12], or a Prolog implementation of the Constraint Handling Rules language CHR [24], should be portable as libraries in a modular Prolog system. Most of the current modular Prolog systems, such as SICStus [25], SWI [29], ECLiPSe [2], XSB [22], Ciao [4, 7, 6] for instance, fall into this category. We will focus on this approach in this paper, together with the *object-oriented approach* [19, 20] which is somewhat similar.

The *algebraic approach* defines module calculi with operations over sets of program clauses [21, 5, 23]. They are somehow more general than the object-oriented extensions of Prolog, as they consider a great variety of operations on predicate definitions, like overriding, merging, etc. On the other hand, the greater versatility does not facilitate reasoning on large programs, and this approach has not been widely adopted.

The *logical approach* to module systems extends the underlying logic of programs. One can cite extensions with nested implications [16, 17], meta-logic [3] or second order predicates [9]. Such logical modules can be created dynamically, similarly to other approaches such as Contextual Logic Programming [18, 1]. Perhaps because their poor compatibility with existing code, they are also not widely used however, and will not be considered in this paper.

2 Review of existing syntactic module systems

In this section, we analyze the main syntactic module systems developed for Prolog. A reference example will be used to illustrate their peculiarities, and classify them according to the two previously introduced properties: the called module code protection and the calling module code protection.

Following ISO Prolog terminology [14], a *module* is a set of Prolog clauses associated to a unique identifier, the *module name*. The *calling context* – or simply *context* – is the name of the module from where a call is made. A *qualified goal* `M:G` is a classical Prolog goal `G` prefixed by a module name `M` in which it must be interpreted. A predicate is *visible* from some context if it can be called

from this particular context without any qualification. A predicate is *accessible* from some context if it can be called from this particular context with or without qualification. A *meta-predicate* is a predicate that handles arguments to be interpreted as goals. Those arguments are called *meta-arguments*.

2.1 A Basic Module System

We first consider a basic module system from which the syntactic module systems of Prolog can be derived through different extensions.

In this basic system, the predicates that are *visible* in a module are either defined in the module, or *imported* from another module. In order to ensure the protection of the called module code, only the predicates explicitly *exported* in the defining module can be imported by other modules. Furthermore, the qualification of predicates is not allowed.

The basic module system thus trivially satisfies both forms of code protection properties, but is not able to modularize the predicate `forall` of example 1.

2.2 SICStus Prolog

The modules of SICStus Prolog [25] make accessible any predicate, by using qualifications. The list iterator *forall* can thus be modularized, and used simply by passing to it goals qualified with the calling module. As a consequence however, this versatile module system does not ensure any form of module code protection.

It is also possible to explicitly declare meta-predicates and meta-arguments. In that case, the non-qualified meta-arguments are qualified dynamically with the calling context of the meta-predicate. With this feature, the called module is thus able to manipulate explicitly the name of the calling module and call any predicate in the calling module.

Example 3. This example, that will be also used in the following, tests the capabilities of calling private predicates in modules.

```
:-module(library, [mycall/1]).

p :-
  write('library:p/0  ').

:-meta_predicate(mycall(:)).
mycall(M:G):-
  M:p, call(M:G).
```

```
:- module(using, [test/0]).
:- use_module(library).

p :-
  write('using:p/0  ').
q :-
  write('using:q/0  ').
test :-
  library:p, mycall(q).
```

```
| ?- using:test.
library:p/0  using:p/0  using:q/0
yes
```

The private predicate p of the library is called from the using module, the library correctly calls the predicate q of the calling module, but is also able to call the private predicate p of the calling module.

This module system is similar to the ones of Quintus Prolog [26] and Yap Prolog [27]. The standardization attempt of ISO-Prolog [14] is also very close in spirit, but the accessibility rules of qualified predicates have been left to the implementation.

2.3 ECLiPSe

ECLiPSe [2] introduces two mechanisms to call non visible predicates. The first is the qualified call, where only the exported predicates are accessible. The second one, which uses the construct `call(Goal)@Module`, makes any predicate accessible as with a qualified goal `Module:Goal` in SICStus Prolog. This system provides also a directive `tool/2` for adding the calling context as an extra argument to the meta-predicate. This solution has the advantage of limiting the unauthorized calls made in a unconscious way.

Example 4.

```
:- module(library, [mycall/1]).

p :-
  write('library:p/0  ').

:- tool(mycall/1,mycall/2).
mycall(G, M):-
  call(p)@M, call(G)@M.
```

```
:- module(using, [test/0]).
:- use_module(library).

p :-
  write('using:p/0  ').
q :-
  write('using:q/0  ').
test :-
  call(p)@library, mycall(q).
```

```
[eclipse 2]: using:test.
library:p/0  using:p/0  using:q/0
Yes
```

As beforehand, the system does not ensure module code protection.

2.4 SWI Prolog

For compatibility reasons, SWI accepts qualified predicates and uses the same policy as SICStus Prolog. Hence the complete system does not ensure the called module code protection. Meta-programming in SWI Prolog [29] has a slightly different semantics. For a meta-call made in the clause of a meta-predicate declared with the directive `module_transparent/1`, the calling context is the calling context of the goal that invoked the meta-predicate. Hence, by declaring the list

iterator `forall/2` as a module transparent predicate, one obtains the expected behavior, since the meta-call to `G` is invoked in the module that called `forall`, i.e. in the calling module.

Nonetheless, this choice has two main disadvantages:

Example 5.

```
:-module(library, [mycall/1]).

p :-
  write('library:p/0  ').

:-module_transparent(mycall/1).
mycall(G):-
  p, call(p), call(G).
```

```
:- module(using, [test/0]).
:- use_module(library).

p :-
  write('using:p/0  ').
q :-
  write('using:q/0  ').
test :-
  mycall(q).
```

```
?- using:test.
library:p/0  using:p/0  using:q/0
Yes
```

First, a dynamic call `call(p(x))` does not behave as the static one `p(x)`. Second, the conventions for meta-predicates break the protection of the calling module code.

2.5 Logtalk

Logtalk [19, 20] is not really a syntactic module system but an object-oriented extension of Prolog. Nonetheless by restricting its syntax – by forbidding parameterized objects and inheritance – one obtains a module system close to the ones studied here.

The `object/1` directive can be read as a `module/2` directive, where the public predicates are the exported predicates. Then, message sending plays the role of goal qualification. Indeed, sending the message `P` to the object `M` – which is denoted by `M::P` instead of `M:P` – calls the predicate `P` defined in the module `M`, only if `P` have been declared public in `M`. Therefore this module system ensures the protection of the called module code.

In order to deal with meta-predicates, Logtalk provides its own version of the `meta_predicate/1` directive, which can be used in a similar way to the SICStus one, with `::` used instead of `:` for declaring meta-argument. As SWI, Logtalk does not realize a module name expansion of the meta-arguments, but realize dynamic calls in a context which may be different from a static call. In this system, the dynamic context is the module (i.e. object) that sent the last message. Since the non qualified calls are not considered as messages however, it is possible to call any predicate of the calling module.

Example 6.

```
:- object(library).
:- public(mycall/1).

p :-
  write('library:p/0  ').

mycall(G) :- mycall(G,p).
:-metapredicate(mycall(::,::)).
mycall(G1,G2) :-
  call(G1), call(G2).

:-end_object.
```

```
:- object(using).
:- public(test/0).

p :-
  write('using:p/0  ').
q :-
  write('using:q/0  ').
test :-
  library::mycall(q).

:- end_object.
```

```
| ?- using::test.
using:q/0  using:p/0
yes
```

That module system does not ensure the calling module protection.

2.6 Ciao Prolog.

The module system of Ciao Prolog [4] satisfies the two forms of code protection. Only exported predicates are accessible for outside a module, and this property is checked for qualified goals. The manipulation of meta-data through the module system is possible through an advanced version of the `meta_predicate/1` directive.

Before calling the meta-predicates, the system dynamically translates the meta-arguments into an internal representation containing the goal and the context in which the goal must be called. Since this translation is done before calling the meta-predicate, the system correctly selects the context in which the meta-data must be called. As far as the system does not document any predicate able to create or manipulate the internal data, the protection of the code is preserved. In this sense, Ciao Prolog does make a distinction between terms and higher-order data (i.e. goals manipulated as terms) [8].

Example 7.

```
:-module(library, [mycall/1]).

p :- write('library:p/0  ').

:-meta_predicate(mycall(:)).
mycall(G):-
  writeq(G), write(' '), call(G).
```

```
:- module(using, [test/0]).
:- use_module(library).

p :- write('using:p/0  ').

test :- mycall(p).
```

```
?-  using:test.
$:('using:p') using:p/0
yes
```

The program realizes the expected behavior without compromising the called module protection, nor the calling module protection.

2.7 XSB

The module system of XSB [22] is an atom-based, rather than predicate-based, syntactic module system. This means that function symbols, as well as predicate symbols, are modularized in XSB. Similar terms constructed in different modules may thus not unify. In a module, it is possible however to import public symbols from another module, with the effect that the modules share the same symbols.

Then, the semantics of the `call/1` predicate is very simple: the meta-call of a term corresponds to the call of the predicate of the same symbol and arity as the module where the term has been created. The system fully satisfies the code protection property.

Example 8.

```
:-export mycall/1.

p(_) :-
  write('library:p/1  ').

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

```
:- export test/1.
:- import mycall/1 from library.

p(_) :-
  write('using:p/0  ').
test(_) :-
  mycall(p(_)).
```

```
| ?-  test(_).
using:p/0
yes
```

On the other hand, the problem of defining the visibility rules for meta-programming predicates is moved to the construction of the terms. Indeed, in XSB, the terms constructed with `=../2`, `functor/2` and `read/1` belong to the module `user`. As a consequence, in a module different from `user`, the goal `(functor(X,q,1), X=q(_))` fails, whereas `(X=q(_), functor(X,q,1))` succeeds.

3 A safe module system with calls and closures

In this section, we define a formal module system with calls and closures. We present the operational semantics of modular logic programs, and formally prove that they satisfy both forms of module code protection.

3.1 Syntax of Modular Logic Programs

For the sake of simplicity of the presentation, the following conventions are adopted. First, a simple form of closures, that abstract only one argument in an atom, is considered. Second, the syntax of constraint logic programs is chosen with some syntactic conventions to distinguish between the constraints, the closures and the other atoms within goals. Third, all goals are assumed to be explicitly qualified, thereby eliminating the need to describe the conventions used for automatically prefixing the non-qualified atoms in a clause or a goal. Fourth, all public predicates in a module are assumed to be accessible from outside, with no consideration of directives such as `use_module`.

The following disjoint alphabets of symbols given with their arity are considered:

- V a countable set of variables (of arity 0) denoted by $x, y \ldots$;
- Σ_F a set of constant (of arity 0) and function symbols;
- Σ_C a set of constraint predicate symbols containing $= /2$ and $true/0$;
- Σ_P a set of program predicate symbols containing $call/2$, $closure/3$ and $apply/2$;
- Σ_M a set of module names (of arity 0), noted $\mu, \nu \ldots$

Furthermore, in order to interpret *calls* and *closures*, two coercion relations, $\overset{P}{\sim}$: $\Sigma_F \times \Sigma_P$ and $\overset{M}{\sim}$: $\Sigma_F \times \Sigma_M$, are assumed to interpret function symbols as predicate symbols and module names respectively. It is worth noting that in classical Prolog systems, where function symbols are not distinguished from predicate symbols, these relations are just the identity, while here they formally relate disjoint sets.

The sets of terms, formed over V and Σ_F, of atomic constraints, formed with predicate symbols in Σ_C and terms, and of atoms, formed with predicate symbols in Σ_P and terms, are defined as usual. In addition, atoms qualified by a module name are noted $\mu{:}A$. The `call` predicate has two arguments, the first being the module name qualifying the second argument.

A closure $closure(x, \mu{:}A, z)$ associates to the variable z a qualified atom $\mu{:}A$ (the meta-argument) in which the variable x is abstracted. The meta-argument in a closure must be a qualified atom, i.e. not a variable as in a *call*.

Definition 1. *A **closure** is an atom of the form $closure(x, \mu : A, z)$ where x and z are variables, $\mu : A$ is a qualified atom. The **application** of a closure associated to a variable z to an argument x is the atom $apply(z, x)$.*

Definition 2. *A **modular clause** is a formula of the form*

$$A_0 \leftarrow c_1, \ldots, c_l | \kappa_1, \ldots, \kappa_n | \mu_1 : A_1, \ldots, \mu_m : A_m.$$

where the c_i's are atomic constraints, the κ_i's are closures, and the $\mu_i : A_i$'s are qualified atoms.

Definition 3. *A **module** is a tuple* $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)$ *where* $\mu \in \Sigma_M$ *is the* name *of the module,* $\mathcal{D}_\mu$ *is a set of clauses, called the* implementation *of the module, and* $\mathcal{I}_\mu \subset \Sigma_P$ *is the set of* public *predicates, called the* interface *of the module. The predicates not in* I_μ *are called* private *in* μ*. A **modular program** $\mathcal{P}$ is a set of modules with distinct names.*

Definition 4. *A **modular goal** is a formula*

$$c|\left\langle \nu_1 - \kappa_1 \right\rangle, \ldots, \left\langle \nu_n - \kappa_n \right\rangle | \left\langle \nu'_1 - \mu_1 : A_1 \right\rangle, \ldots, \left\langle \nu'_m - \mu_m : A_m \right\rangle$$

where c *is a set of atomic constraints, the* κ_i*'s are closures, the* $(\mu_i : A_i)$*'s are prefixed atoms and both the* ν_i*'s and the* ν'_i*'s are module names called* calling contexts.

In the following, the construct $\langle \nu - (\kappa_1, \ldots, \kappa_n) \rangle$ denotes the sequence of closures $(\langle \nu - \kappa_1 \rangle, \ldots, \langle \nu - \kappa_n \rangle)$ and similarly for sequence of atoms with context.

3.2 Operational Semantics

Let $\mathcal{P}$ be a program defined over some constraint system $\mathcal{X}$. The transition relation $\longrightarrow$ on goals is defined as the least relation satisfying the rules in table 1, where θ is a renaming substitution with fresh variables. A successful derivation for a goal G is a finite sequence of transitions from G which ends with a goal containing only constraints (the computed answer) and closures.

Modular CSLD

$$\frac{(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \qquad (\nu = \mu) \vee (p \in \mathcal{I}_\mu) \qquad (p(\boldsymbol{s}) \leftarrow c' | k | \beta)\theta \in \mathcal{D}_\mu \qquad \mathcal{X} \models \exists (c \wedge \boldsymbol{s} = \boldsymbol{t} \wedge c')}{(c|K|\gamma, \langle \nu - \mu : p(\boldsymbol{t}) \rangle, \gamma') \longrightarrow (c, \boldsymbol{s} = \boldsymbol{t}, c' | K, \langle \mu - k \rangle | \gamma, \langle \mu - \beta \rangle, \gamma')}$$

Call

$$\frac{\mathcal{X} \models c \Rightarrow (s = g \wedge t = f(\boldsymbol{x})) \qquad g \overset{M}{\sim} \mu \qquad f \overset{P}{\sim} p}{(c|K|\gamma, \langle \nu - \nu : call(s,t) \rangle, \gamma') \longrightarrow (c, s = g, t = f(\boldsymbol{x}) | K | \gamma, \langle \nu - \mu : p(\boldsymbol{x}) \rangle, \gamma')}$$

Apply

$$\frac{\mathcal{X} \models c \Rightarrow z = y}{\begin{array}{c}(c|\kappa_1, \langle \mu - closure(x, \mu' : A, z) \rangle, \kappa_2 | \gamma, \langle \nu - \nu : apply(y,t) \rangle, \gamma') \longrightarrow \\ (c|\kappa_1, \langle \mu - closure(x, \mu' : A, z) \rangle, \kappa_2 | \gamma, \langle \nu - \mu' : A[x \backslash t] \rangle, \gamma')\end{array}}$$

Table 1. Transition relation for goals with calls and closures.

The *modular CSLD* resolution rule is a restriction of the classical CSLD rule for CLP [15]. The additional condition $(\nu = \mu) \vee (p \in \mathcal{I}_\mu)$ imposes that $\mu : p(\boldsymbol{t})$ can be executed only if, either the call is made from inside the module (i.e. from the calling context μ), or the predicate p is a public predicate in μ. Moreover, this rule propagates the new calling context to the closures and atoms of the selected clause body.

The *call* rule defines the operational semantics of meta-calls. It is worth noting that this transition rule does not change the calling context ν. This property is necessary to guarantee the calling module code protection. For the sake of simplicity, a *call* goal with a free variable as meta-argument has no transition. Similarly, the *call* rule does not handle the meta-call of conjunctions of atoms or constraints. Those meta-calls can nevertheless be easily emulated, by supposing $(',')/2 \overset{P}{\sim} and/2)$ and by adding the clause $(and(x, y) \leftarrow \mu : call(x), \mu : call(y))$ to the implementation of any module μ.

The *apply* rule allows the invocation of a closure collected by a previous predicate call, as expected for instance in the example 2 for the definition of `forall`. In practice, the *apply* rule looks for the closure associated to the closure variable (formally checks the equality of variables $z = y$), and applies the closure to the argument in the closure context.

3.3 Module Code Protection

Intuitively, the called module code protection property states that only the public predicates of a module μ can be called from outside, and produce subgoals in context μ. The calling module code protection property states that the goal of a closure can only be executed in the context of creation of the closure. These properties can be formalized as follows:

Definition 5. *The operational semantics of programs satisfies the* **called module code protection** *if the reduction of a qualified atom $\mu : p(\boldsymbol{t})$ in a context ν produces qualified atoms and closures in the context μ only, and either p is public in μ or $\mu = \nu$.*

Definition 6. *The operational semantics of programs satisfies the* **calling module code protection** *property if the application of a closure created in context ν produces atoms and closures in the context ν only.*

Proposition 1. *The operational semantics of modular logic programs satisfies the called and calling module code protection properties.*

Proof. For the called module code protection property, let us consider the reduction of a qualified atom $\mu : p(\boldsymbol{t})$ in context ν. Only a modular CSLD or a call transition can apply, producing a goal in context μ'. In the former case, we have $\mu' = \mu$ and either $\mu = \nu$ or p public in μ. In the latter case, we have trivially $\mu = \nu = \nu'$.

For the calling module code protection property, we first remark that the transition rules do not change the context of closures, which thus remain in their context of creation. Given an application of a closure created in context ν, the transition **Apply** is the only applicable rule, and produces a goal in context ν.

4 Logical Semantics

Syntactic module systems have been criticized for their lack of logical semantics [21, 23]. Here we provide modular (constraint) logic programs without calls nor closures (abbreviated MCLP), with a logical semantics based on their translation into constraint logic programs. In course, that translation describes an implementation of the module system.

To a given MCLP program $\mathcal{P}$, one can associate a simple module constraint system $\mathcal{M}$, in which the constraint $allow(\nu, \mu, p)$ that states that the predicate p of module μ can be called in module ν, is defined by the following axiom schemas:

$$\frac{\nu \in \Sigma_M \quad p \in \Sigma_P}{\mathcal{M} \models allow(\nu, \nu, p)} \qquad \frac{\nu, \mu \in \Sigma_M \quad (\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \quad p \in \mathcal{I}_\mu}{\mathcal{M} \models allow(\nu, \mu, p)}$$

This constraint system depends solely on the interface of the different modules that composes the program $\mathcal{P}$, and not on its implementation.

Then, MCLP programs can be given a logical semantics equivalent to their operational semantics, obtained by a simple translation of pure MCLP($\mathcal{X}$) programs into ordinary CLP($\mathcal{M}, \mathcal{X}$) programs. This translation can be used for the implementation, and shows that the module system can be viewed as simple syntactic sugar. The alphabet $\dot{\Sigma}_P$ of the associated CLP($\mathcal{M}, \mathcal{X}$) program, is constructed by associating one and only one predicate symbol $\dot{p} \in \dot{\Sigma}_P$ of arity $n + 2$ to each predicate symbol $p \in \Sigma_P$ of arity n.

Let Π be the translation of MCLP programs and goals into CLP programs over $\mathcal{M}$, defined in table 2.

$$\begin{array}{l}
\Pi\left(\bigcup\{(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)\}\right) = \bigcup\{\Pi_\mu(\mathcal{D}_\mu)\} \\
\Pi\left(\gamma, \gamma'\right) = \Pi\left(\gamma\right), \Pi\left(\gamma'\right) \\
\Pi\left(\langle \nu - \mu : p(\boldsymbol{t})\rangle\right) = \dot{p}(\nu, \mu, \boldsymbol{t}) \\
\\
\Pi_\mu\left(\bigcup\{(A \leftarrow c|\alpha)\}\right) = \bigcup\{\Pi_\mu(A \leftarrow c|\alpha)\} \\
\Pi_\mu\left(p_0(\boldsymbol{t}) \leftarrow c|\alpha\right) = \dot{p_0}(y, \mu, \boldsymbol{t}) \leftarrow allow(\mu, y, p_0), c|\Pi_\mu(\alpha) \\
\Pi_\mu(A, A') = \Pi_\mu(A), \Pi_\mu(A') \\
\Pi_\mu(\nu : p(\boldsymbol{t})) = \dot{p}(\mu, \nu, \boldsymbol{t})
\end{array}$$

Table 2. Formal translation of MCLP($\mathcal{X}$) into CLP($\mathcal{M}, \mathcal{X}$).

This translation basically adds two arguments to each predicate. The first argument is the calling context and the second is the qualification. The constraint *allow* realizes a dynamic check of accessibility. It is worth noting that for a qualified atom, the contexts are known at compile-time and the accessibility check can be done statically, thereby eliminating any overhead due to the added

constraints and to the module system. On the other hand, for the *call* predicate not considered in this section, the *allow* predicate implements a dynamic check, hence with an overhead due to the added constraints.

Proposition 2 (Soundness). *Let $\mathcal{P}$ and $(c|\gamma)$ be a pure MCLP program and a pure MCLP goal*

$$if \quad \left((c|\gamma) \xrightarrow[MCLP]{\mathcal{P}} (d|\gamma')\right) then \left((c|\Pi(\gamma)) \xrightarrow[CLP]{\Pi(\mathcal{P})} (d, allow(y,\mu,p), y=\nu|\Pi(\gamma'))\right)$$

for some ν, μ, p and some y is not free in d.

Proof. Let us suppose $((c|\gamma) \xrightarrow[MCLP]{\mathcal{P}} (d|\gamma'))$. Let $\langle \nu - \mu : p(\boldsymbol{t})\rangle$ be the selected atom in γ. Then γ is of the form $(\gamma_1, \langle \nu - \mu : p(\boldsymbol{t})\rangle, \gamma_2)$ for some γ_1 and γ_2. Hence we have $\Pi(\gamma) = (\Pi(\gamma_1), \dot{p}(\nu,\mu,\boldsymbol{t}), \Pi(\gamma_2))$. Now let $(p(\boldsymbol{s}) \leftarrow c'|\alpha)\theta$ be the selected clause in module μ. In such a case whe have, in the translation of $\mathcal{P}$, the clause $(\dot{p}(y,\mu,\boldsymbol{s})) \leftarrow c, allow(y,\mu,p)|\Pi_\mu(\alpha))\,\theta$. We also have $d = (c, \boldsymbol{t}=\boldsymbol{s}, c')$, $X \models \exists(d)$ and $(\nu = \mu) \vee (p \in \mathcal{I})$. As $(\nu = \mu) \vee (p \in \mathcal{I})$ is true, the constraint $allow(\nu,\mu,p)$ is true in $\mathcal{M}$, hence we have $\mathcal{X}, \mathcal{M} \models \exists d'$ with $d' = (c, (\nu,\mu,\boldsymbol{t}) = (y,\mu,\boldsymbol{s}), c', allow(y,\mu,p))$. Therefore we have $((c|\Pi(\gamma)) \xrightarrow[CLP]{\Pi(\mathcal{P})} (d'|\Pi(\gamma')))$.

Lemma 1. *The functions Π_μ, and Π on goals, are injective.*

Proof. As it is the composition of injective functions, the function Π on goals is injective. For the same reason, the function Π_μ on prefixed atoms, atom sequences and clauses is injective. As Π_μ on modules is the pointwise extension of the injective function Π_μ on clauses, it is injective too.

Proposition 3 (Completeness). *Let $\mathcal{P}$ and $(c|\gamma)$ be pure MCLP program and goal*

$$if \quad \left((c|\Pi(\gamma')) \xrightarrow[CLP]{\Pi(\mathcal{P})} (d|\alpha)\right) \ then \ \left((c|\gamma) \xrightarrow[MCLP]{\mathcal{P}} (d'|\gamma'')\right)$$

where $\Pi(\gamma'') = \gamma'$ and $d' = (d, allow(y,\mu,p), y = \nu)$ for some ν, μ, p and such that y is not free in d.

Proof. Because Π_μ and Π are injective, we can use their respective inverses ${\Pi_\mu}^{-1}$ and Π^{-1}. Let us suppose that $((c|\Pi(\gamma)) \xrightarrow[CLP]{\Pi(\mathcal{P})} (d|\gamma'))$. The constraint c does not contain any $allow/3$ constraint since $(c|\gamma)$ is a MCLP goal. Let $q(\boldsymbol{t})$ be the selected atom, $\Pi(\gamma)$ is of the form $(\gamma_1, q(\boldsymbol{t}), \gamma_2)$ for some γ_1 and γ_2. Hence we have $\gamma = \Pi^{-1}(\gamma_1), p(\nu,\mu,\boldsymbol{t'}), \Pi^{-1}(\gamma_2)$ with $q = \dot{p}$ and $t = (\nu,\mu,\boldsymbol{t'})$. Let $q(\boldsymbol{s}) \leftarrow c'|\beta$ be the selected clause. We have $p(\boldsymbol{s'}) \leftarrow c''|{\Pi_{\mu'}}^{-1}(\beta)$ in the implementation of some module μ', with $\boldsymbol{s} = (y,\mu',\boldsymbol{s'})$ and $c'' = c', allow(y,\mu',p)$ where y is fresh. We have $d = (c, c'', allow(y,\mu',p), (\nu,\mu,\boldsymbol{t'}) = (y,\mu',\boldsymbol{s'}))$ and $\mathcal{X}, \mathcal{M} \models \exists(d)$. Hence for $d' = (c, c', \boldsymbol{t'} = \boldsymbol{s'})$, we have $\mathcal{X}, \mathcal{M} \models \exists(d')$. Therefore, for $\alpha = (\Pi^{-1}(\gamma_1), \Pi^{-1}(\beta), \Pi^{-1}(\gamma_2))$, we conclude that $(c|\gamma)$ can be reduced by a clause of $\mathcal{P}$ to $(d'|\gamma'')$ with $\Pi(\gamma'') = \gamma'$.

5 Conclusion

In a paper of D.H.D. Warren [28], the higher-order extensions of Prolog were questioned as they do not really provide more expressive power than meta-programming predicates. We have shown here that the situation is different in the context of modular logic programming, and that module code protection issues necessitate to distinguish between calls and closures.

The module system we propose is close to the one of Ciao Prolog in its implementation. We gave an operational semantics for modular logic programs with calls and closures , and used it to formally prove the full module code protection property. Furthermore, an equivalent logical semantics for modular logic programs without calls nor closures has been provided, showing a translation of modular logic programs into constraint logic programs. The logical semantics of modular calls and closures is currently under investigation in the framework of linear logic concurrent constraint (LCC) programming [10].

This module system has been implemented in GNU-Prolog, and has been used to port some existing code as libraries. The modularization of the Constraint Handling Rules language CHR obtained by porting a Prolog implementation [24] as a library, provides an interesting example of intensive use of the module system, as it allows the development of several layers of constraint solvers in CHR. New libraries are also developed with this module system for making a fully bootstrapped implementation of the LCC language SiLCC [11].

Acknowledgements. We are grateful to Sumit Kumar for a preliminary work he did on this topic, during his summer 2003 internship at INRIA. We thank also Emmanuel Coquery and Sylvain Soliman for valuable discussion on the subject. and Daniel de Rauglaudre and Olivier Bouissou for their comments and help in the implementation.

References

1. S. Abreu and D. Diaz. Objective: in minimum context. In *Proceedings of ICLP'2003, International Conference on Logic Programming*, Mumbai, India, 2003. MIT Press.
2. A. Aggoun and al. *ECLiPSe User Manual Release 5.2*, 1993 – 2001.
3. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In *META-92: Third International Workshop on Meta Programming in Logic*, pages 105–119, Berlin, Heidelberg, 1992. Springer-Verlag.
4. F. Bueno, D. C. Gras, M. Carro, M. V. Hermenegildo, P. Lopez-Garca, and G. Puebla. The ciao Prolog system. reference manual. Technical Report CLIP 3/97-1.10#5, University of Madrid, 1997-2004.
5. M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programmming*, 19/20:443–502, 1994.
6. D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid, Aug. 2004.
7. D. Cabeza and M. Hermenegildo. A new module system for Prolog. In *First International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 131–148. Springer-Verlag, July 2000.

8. D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A type-free higher-order logic programming langugae with predicate abstraction. In *Proceedings of ASIAN'04, Asian Computing Science Conference*, pages 93–108, Chiang Mai, 2004. Springer-Verlag.
9. W. Chen. A theory of modules based on second-order logic. In *The fourth IEEE. Internatal Symposium on Logic Programming*, pages 24–33, 1987.
10. F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, Feb. 2001.
11. R. Haemmerlé. SiLCC is linear concurrent constraint programming (doctoral consortium). In *Proceedings of International Conference on Logic Programming ICLP 2005*, Lecture Notes in Computer Science. Springer-Verlag, 2005.
12. C. Holzbaur. Oefai clp(q,r) manual rev. 1.3.2. Technical Report TR-95-09, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1995.
13. International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 1: General core*, 1995. ISO/IEC 13211-1.
14. International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 2: Modules*, 2000. ISO/IEC 13211-2.
15. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, Jan. 1987.
16. D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.
17. D. Miller. A proposal for modules in lambda prolog. In *Proceedings of the 1993 Workshop on Extensions to Logic Programming*, volume 798 of *Lecture Notes in Computer Science*, pages 206–221, 1994.
18. L. Monterio and A. Porto. Contextual logic programming. In *Proceedings of ICLP'1989,International Conference on Logic Programming*, pages 284–299, 1989.
19. P. Moura. Logtalk. `http//www.logtalk.org`.
20. P. Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Informatics, University of Beira Interior, Portugal, Sept. 2003.
21. R. A. O'Keefe. Towards an algebra for constructing logic programs. In *Symposium on Logic Programming*, pages 152–160. IEEE, 1985.
22. K. Sagonas and al. *The XSB System Version 2.5 - Volume 1: Programmer's Manual*, 1993 – 2003.
23. D. T. Sannella and L. A. Wallen. a calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, pages 147–177, 1992.
24. T. Schrijvers and D. S. Warren. Constraint handling rules and table execution. In *Proceedings of ICLP'04, International Conference on Logic Programming*, pages 120–136, Saint-Malo, 2004. Springer-Verlag.
25. Swedish Institute of Computer Science. *SICStus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 1991–2004.
26. Swedish Institute of Computer Science. *Quintus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 2003.
27. R. R. Víctor Santos Costa, Luís Damas and R. A. Diaz. *YAP user's manual*, 1989–2000.
28. D. H. D. Warren. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence*, volume 10 of *Lecture Notes in Mathematics*, pages 441–454. 1982.
29. J. Wielemaker. *SWI Prolog 5.4.1 Reference Manual*, 1990– 2004.