Closures are Needed for Closed Module Systems

Rémy Haemmerlé, Francois Fages

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Closures are Needed for Closed Module Systems

Rémy Haemmerlé — François Fages

N° 5575

Mai 2005

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5575--FR+ENG

Closures are Needed for Closed Module Systems

Rémy Haemmerlé , François Fages

Thème SYM — Systèmes symboliques
Projets Contraintes

Rapport de recherche n° 5575 — Mai 2005 — 19 pages

Abstract: In a classical paper of D.H.D. Warren, the higher-order extensions of Prolog were questioned as they do not really provide more expressive power than meta-programming predicates. Without disputing this argumentation in the context of a logic programming system without modules, we show that the situation is different in a closed module system. By *closed* we mean the property that the module system is able to prevent any call to the private predicates of a module from the other modules, in particular through meta-programming predicates. We show that this property necessitates to distinguish the execution of a term (meta-programming predicate `call`) from the execution of a closure (higher order). We propose a module system for Constraint Logic Programming with a notion of closures inspired from Linear Concurrent Constraint programming. This module system is quite simple and pretty independent of a precise language (it is currently implemented for GNU-Prolog). Although this system can be seen as a simple layer of syntactic sugar, it does provide a discipline for naming predicates and hiding code, making possible the development of libraries and facilitating the safe re-use of existing code. Furthermore we provide the module system with logical and operational semantics. This formal setting is used in the paper to compare our approach to the other module systems proposed for Prolog, which generally do not ensure full code protection.

Key-words: constraint logic programming, module system, closures, higher-order programming, meta-programming.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Les fermetures sont nécessaires aux systemes de modules fermés

Résumé : Dans un de ses articles D.H.D. Warren critiquait les extensions de Prolog par des mécanismes d'ordre supérieur (notamment les fermetures). Il montre, en effet, que le langage obtenu n'est pas réellement plus expressif qu'un Prolog avec méta-programmation seule. Même si nous ne remettons pas en cause cette argumentation dans le cadre d'un système sans module, nous montrons que la situation est différente dans le cas d'un système de modules fermé. Par *fermé* nous entendons, le fait qu'un système de modules doit être capable d'empêcher tout appel aux parties privées d'un module depuis l'extérieur de celui-ci. Nous montrons que cette propriété nécessite de distinguer l'exécution d'un terme (c.à.d. l'utilisation du prédicat de méta-prorgammation `call`) de l'exécution d'une fermeture (c.à.d. l'utilisation de l'ordre supérieur). Nous proposons un système de modules pour la programmation logique avec contraintes étendue avec une notion de fermeture inspirée de la programmation linéaire concurrente avec contraintes. Ce système de modules est assez simple et relativement indépendant d'un langage particulier (il est actuellement développé en GNU-Prolog). Bien qu'il puisse être vu comme une simple extension syntaxique, ce système permet une discipline de nommage des prédicats permettant de développer des bibliothèques et de faciliter la réutilisation du code. Dans cet article nous donnons à notre système une sémantique opérationnelle et une sémantique logique. Cette présentation formelle est utilisée pour comparer notre approche aux autres systèmes de modules pour Prolog qui de façon générale ne garantissent pas la protection du code.

Mots-clés : programmation logique avec contraintes, système de modules, fermetures, programmation d'ordre supérieur, meta-programmation.

1 Introduction

It is often easier to work with a small number of concepts in mind. Thus to be written, large programs must be divided into small parts. Although this can be done in any language, a suitable module system helps the segmentation of programs by forcing the programmer to clearly define the limits and the interface of each block. Hence many errors can be avoided, the compiler being in charge of verifying that the calls between modules respect the declared interfaces. Moreover such divisions facilitate the understanding of a program by external programmers and improve the reusability of the code.

The context of our work is the design of a fully bootstrapped implementation of a Linear logic Concurrent Constraint (LCC) [11, 25] programming language called **SiLCC**, for "SiLCC is Linear Concurrent Constraint programming". The purpose of the present paper is to present the theoretical and practical choices we made for its module system. For the sake of simplicity and because we think, as Leroy [18], that "modular programming has little to do with the particulars of any programming language" we will present our module system in the formal setting of Constraint Logic Programs (CLP) [17] rather than in the more general setting of LCC.

The purpose of a module system is not unique and it is important to list the different purposes a module system may serve:

Implementation Hiding. A programmer should have the possibility to protect his code from the intrusion from other modules. This means that it should be possible to restrict the access of a module (i.e. calls, dynamic assertions/retractions, syntax modifications, global variable assignments etc ...) from extra-modular code. In the following, we will say that a module system that ensures *code protection* is a *closed* module system, on the contrary a system that allows any call from a module to another will be called *open*.

Separation of name space. In large *flat* (i.e. without modules) system, name clashes are frequent and disturbing. Indeed, to be certain to avoid such clashes, the programmer is constrained to systematically prefix the name of his predicates. A module system has to avoid such clashes automatically.

Separate compilation. It should be possible to compile in an independent way each module of a program. In particular, standard libraries must be compiled once for all.

Bootstrapping. One of the most original aspects of SiLCC is the realization of a completely bootstrapped implementation of a constraint language from a small kernel (LCC with constraints over labeled graphs and linear logic constraints providing imperative features). Modules are essential to the bootstrap of such a complex language.

Furthermore, the design of a module system for Prolog needs to meet some other requirements:

Simplicity. The module system should not restrict the use of Prolog for rapid prototyping, in particular :

- the conciseness of Prolog code should be preserved, avoiding too many module-related declarations;
- meta-programming predicates such as the `call` predicate should be supported;
- new concepts should be limited in order to be adopted by classical Prolog programmers.

Semantics. The module system should come with a formal semantics from which one can derive its properties.

Simple implementation. We want the module system to be easy to code and to port onto different logic languages, ranging from GNU-Prolog to SiLCC.

Modularity in the context of logic programming has been considerably studied, and there has been some standardization attempts [16]. In order to define a notion of module that captures important aspects such as import/export and implementation hiding, the *logical approaches* rely on an extension of the underlying logic. For example, one can cite extensions with nested implications [19], meta-logic [2] or second order predicates [7]. Some other approaches, such as contextual logic programming [20], or object-oriented extensions [21], go even further in the direction of fully dynamic module systems.

Other proposals add constructs for declaring and handling a notion of static modules. On the one hand, there are *algebraic approaches* defining module calculi on sets of program clauses, such as the module calculus of O'Keefe [22], or of Bugliesi, Lamma et Mello [4], or the calculus of Sanella and Wallen [24] inspired from functional programming. On the other hand, the so-called *syntactic approaches* deal mainly with the alphabet of symbols. This approach is criticized in [22, 24] for its lack of logical semantics. Nevertheless syntactic module systems are often chosen for their simplicity and compatibility with existing code. For instance, the existing code of OEFAI CLP(q,r) [14] or a Prolog implementation of CHR [26] should be ported as libraries in a modular system. Most of current modular Prolog systems, such as SICStus [27], SWI [29], ECLiPSe [1], XSB [23], Ciao [3, 6, 5], fall into this category. None of them however ensures every purpose we have exposed previously. For example, the popular system SICStus does not provide any kind of implementation hiding.

In this paper, we show that closures are needed to ensure code protection in a predicate-based syntactic module system. We propose a closed module system which distinguishes meta-programming predicates from closures. We give an operational semantics which is used to prove the properties of the module system, and to compare the different existing systems. Furthermore we provide a logical semantics with a translation of modular constraint logic programs into CLP programs with a simple module constraint system.

The paper is organized as follows. The next section defines the syntax of the module system we propose for CLP. Section 3 presents the operational semantics of modular constraint logic programs (MCLP) with meta-calls and closures, and shows that this module

system satisfies the code protection property. Section 4 presents the logical semantics of pure MCLP programs, i.e without meta-predicates nor closures. Then, in section 5, we discuss some pragmatic aspects of our system. Finally we compare our proposal with existing syntactic module systems and conclude.

2 Modular Constraint Logic Programs

We consider the following disjoint alphabets:

- V a set of countable variables denoted by $x, y \ldots$;
- Σ_F a set of constant and function symbols;
- Σ_C a set of constraint predicate symbols containing $=$ and $true$;
- Σ_P a set of program predicate symbols containing $call/2$, $closure/3$ and $apply/2$;
- Σ_M a set of module names,
- two relations $\overset{M}{\rightsquigarrow}: \Sigma_F \times \Sigma_M$ and $\overset{P}{\rightsquigarrow}: \Sigma_F \times \Sigma_P$

and the usual sets of terms, formed over V and Σ_F, atomic constraints, formed with predicate symbols in Σ_C, and atoms, formed with predicate symbols in Σ_P. In addition, we consider module names, noted μ,..., and atoms prefixed with a module name, noted $\mu : A$ and called *qualified atoms*.

Because Σ_F, Σ_P and Σ_M are supposed to be distinct, the two relations $\overset{P}{\rightsquigarrow}$ and $\overset{M}{\rightsquigarrow}$ will be useful to interpret function symbols respectively as predicates symbols and as module names while a meta-call. In classical Prolog systems, where function symbols, predicate symbols and module names are not syntactically distinguished, this two relations can be viewed as the trivial bijections.

For the sake of simplicity, we assume here that all atoms are qualified and do not describe the standard conventions (given in section 5) that are used for prefixing automatically the atoms given without module names in a clause or a goal.

Definition 2.1 *A **closure** is a formula of the form*

$$closure(x, \mu : A, z)$$

where x and z are variables, A is an atom and μ is a module identifier.

The closure $closure(x, \mu : A, z)$ associates to variable z a qualified atom $\mu : A$ in which the variable x is abstracted. A closure z is applied to an argument x with the predicate $apply(z, x)$.

Definition 2.2 *A **MCLP 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 κ_i's are closures, the A_i's are atoms, the c_i's are atomic constraints and the μ_i's are module names.

Definition 2.3 *A **module** is a tuple $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)$ where $\mu \in \Sigma_M$ is the* name *of the module, $\mathcal{D}_\mu$ a set of clauses is called* implementation *of the module and $\mathcal{I}_\mu \subset \Sigma_P$ is the* interface *of the module. If a symbol p belongs to the interface of a module called μ, we will say that p is* public *in μ, otherwise we will say that p is* private *in μ .*

Definition 2.4 *A **MCLP Program** $\mathcal{P}$ is a set of modules whose names are distinct.*

Definition 2.5 *A **MCLP goal** is a formula*

$$c_1, \ldots, c_l | \langle \nu_1 - \kappa \rangle, \ldots, \langle \nu_n - \kappa_n \rangle | \langle \nu'_1 - \mu_1 : A_1 \rangle, \ldots, \langle \nu'_m - \mu_m : A_n \rangle$$

where the c_i's are atomic constraints, the κ_i are closures, the $(\mu_i : A_i)$'s are qualified atoms and both the ν_i's and the ν'_i's are module names called in this case calling contexts. *$\langle \nu_i - \mu_i : A_i \rangle$'s are called* atoms with context *whereas $\langle \nu_i - \kappa_i \rangle$'s are called* closures with context.

In the following $\langle \nu - (\kappa_1, \ldots, \kappa_n) \rangle$ will be a notation for the sequences of closures with context $(\langle \nu - \kappa_1 \rangle, \ldots, \langle \nu - \kappa_n \rangle)$ and $\langle \nu - (\mu_1 : A_1, \ldots, \mu_m : A_m) \rangle$ a notation for the sequence of atoms with context $(\langle \nu - \mu_1 : A_1 \rangle, \ldots, \langle \nu - \mu_m : A_m \rangle)$.

Example 2.6 *The following classical implementation of the* `findall/3` *predicate illustrates the difference between a meta-call and the application of a closure in a module.*

```
findall(X,G,_) :- call(G),
                  asserta(found(X)),
                  fail.
findall(_,_,L) :- collect([],L).

collect(S,L) :- retract(found(X)),
                collect([X|S],L).
collect(L,L).
```

The use of `call` *in this implementation does not prevent intrusion from another module with a goal like* `findall(X,retract(found(X)),L)`*. On the other hand, an implementation of* `findall` *using closures and* `apply` *instead of* `call` *as below*

```
findall(C,_) :- apply(X,C),
                asserta(found(X)),
                fail.
findall(_,L) :- collect([],L).
```

ensures the protection of the code. For instance, the clauses retracted by the closure in `closure(X,retract(found(X)),C),findall(C,L)` *will be safely retracted in the calling module, and not in the called module.*

In the reverse direction, a typical use of meta-calls instead of closures is for the writing of a top-level or a meta-interpreter.

3 Operational Semantics

3.1 Transition System

Definition 3.1 *Let $\mathcal{P}$ be a MCLP program. The rewriting relation $\longrightarrow$ on goals is defined as the least relation satisfying the rules in table 1.*

Modular CSLD	$\dfrac{(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \qquad (\nu = \mu) \vee (p \in \mathcal{I}_\mu) \qquad (p(\vec{s}) \leftarrow c' \vert k \vert \beta) \theta \in \mathcal{D}_\mu \qquad \mathcal{X} \models \exists (c \wedge \vec{s} = \vec{t} \wedge c')}{(c \vert K \vert \gamma, \langle \nu - \mu : p(\vec{t}) \rangle, \gamma') \longrightarrow (c, \vec{s} = \vec{t}, c' \vert K, \langle \mu - k \rangle \vert \gamma, \langle \mu - \beta \rangle, \gamma')}$
Call	$\dfrac{\mathcal{X} \models \exists (c \wedge t = g \wedge s = f(\vec{x})) \qquad f \overset{P}{\rightsquigarrow} p \qquad g \overset{M}{\rightsquigarrow} \mu}{(c \vert K \vert \gamma, \langle \nu - \nu : call(t,s) \rangle, \gamma') \longrightarrow (c, t = g, s = f(\vec{x}) \vert K \vert \gamma, \langle \nu - \mu : p(\vec{x}) \rangle, \gamma')}$
Apply	$\dfrac{\langle \mu - closure(x, \mu' : A, z) \rangle \in K \qquad \mathcal{X} \models c \Rightarrow z = y}{(c \vert K \vert \gamma, \langle \nu - \nu : apply(y,t) \rangle, \gamma') \longrightarrow (c \vert K \vert \gamma, \langle \mu - \mu' : A[x \backslash t] \rangle, \gamma')}$

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

3.1.1 Modular CSLD.

The *modular CSLD* resolution rule is a restriction of the classical CSLD rule for CLP [17]. The additional condition $(\nu = \mu) \vee (p \in \mathcal{I}_\mu)$ imposes that $\mu : p(\vec{t})$ 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 calling context to the new atoms of the body of the selected clause.

In the following, we call *pure MCLP* languages, the class of MCLP languages built with the *modular CSLD* rule only. In this class, closures cannot be executed and are forbidden in clauses and goals.

3.1.2 Meta-call.

The *call* rule gives an operational semantics to meta-calls. It translates two terms into a qualified atom, respecting both the conversion relations $\overset{P}{\rightsquigarrow}$ and $\overset{M}{\rightsquigarrow}$. It is worth noting that this rule does not change the calling context, which is necessary to guarantee the implementation hiding.

For the sake of simplicity, our definition of $call/2$ does not allow the meta-call of conjunctions of atoms nor the meta-call of a constraint. These calls can be emulated however, by supposing $(',')/2 \overset{P}{\rightsquigarrow} and/2)$ and by adding the clause $(and(x,y) \leftarrow \mu{:}call(x), \mu{:}call(y))$ to the implementation of any module μ.

In this formal semantics of meta-calls, the goal $(c|\, \langle \mu - \nu{:}call(t,s)\rangle)$ succeeds even if the argument of $call/2$ is a free variable. A failure would not be a better solution however, as it would not preserve the independence of the selection strategy. This can be illustrated by the following goal (`X=true, call(X)`). If the meta-call of a free variable failed, a left-to-right selection strategy would lead to the computed answer `X=true` whereas a right-to-left strategy would lead to a failure. The proper way to handle such errors is to raise an exception, which is not formalized here.

3.1.3 Closure.

The *apply* rule allows the invocation of a closure collected by a previous predicate call. In practice, it 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. One can remark that $closure/3$ and $apply/2$ can be defined in LCC as mere syntactic sugar

$$closure(x,A,z) \equiv !\forall x.((arg(z,x)) \rightarrow A)$$
$$apply(t,z) \equiv arg(z,t)$$

In this definition, a closure is an LCC agent which waits for its argument given in a token (linear logic constraint) $arg(z,x)$ that is posted by the apply agent.

Example 3.2 *Closures can be used to define general iterator predicates for data structures. In a module defining some data structure, it is possible to define the binary predicates* `forall/2` *and* `exists/2` *that check that every (resp. at least one) element of a data structure passed in the first argument, verifies a property passed as a closure in the second argument. For instance, in a library for lists, such iterators can be defined as follows:*

```
forall([], C).
forall([X|T], C) :- apply(C, X), forall(T, Z).

exists([X|_], C) :- apply(C, X).
exists([_|T], C) :- exists(T, C).
```

Example 3.3 *It is instructive to try to employ a meta-call instead of a closure in the previous example.*

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

Now let `foo/1` *be a private predicate defined in a different module from the one defining* `forall/1`*. Then, a goal like* `forall([1,2,3], foo)` *will always fail as* `foo` *is not visible. To bypass this restriction, a naive solution consists of declaring as public every predicate used as high-order data (*`foo/1` *in our example). Because this method violates the protection of the code implied by implementation hiding, it is not acceptable. Indeed the system is no more capable to prevent any call of a predicate used as high-order data.*

3.2 Implementation Hiding

In this section, we propose a formal definition of the implementation hiding property, and show that our module system respects it.

Lemma 3.4 *If* $(c|K|\alpha) \longrightarrow (c'|K'|\beta', \langle \nu - A \rangle, \beta'')$ *is a transition then :*

1. *Either there exists* $\langle \nu - \mu{:}p(\vec{x}) \rangle$ *in* α*;*
2. *or there exists* $\langle \nu - closure(x, B, z) \rangle$ *in* K *with* $A{=}B[x\backslash t]$ *for some* t*;*
3. *or there exists* $\langle \mu - \nu{:}p(\vec{x}) \rangle$ *in* α *such that* p *is public in* ν*.*

Proof. Let us suppose that $\langle \nu - A \rangle$ is not in α, as if it is not the case we are trivially in case *1*. If the transition is a *modular CSLD* transition, α is of the form $\gamma', \langle \nu' - \nu{:}p(\vec{x}) \rangle, \gamma''$ with $(\nu'{=}\nu) \vee (p \in \mathcal{I}_\nu)$. If $(\nu'{=}\nu)$ then we are in case *1*, otherwise $(p \in \mathcal{I}_\nu)$ and we are in case *3*. If the transition is a *call*, α is of the form $\gamma', \langle \nu - \nu{:}call(s,t) \rangle, \gamma''$ which corresponds to case *1*. Finally if the transition is an *apply*, $\langle \nu - closure(x, B, z) \rangle$ is in K with $A = B[x\backslash t]$ for some term t, which corresponds to case *2*. □

Lemma 3.5 *If* $(c_0|K_0|\alpha_0) \longrightarrow (c_1|K_1|\alpha_1) \longrightarrow \ldots \longrightarrow (c_n|K_n|\alpha_n)$ *is a derivation and if* $\langle \nu - \kappa \rangle$ *is a closure with context in the sequence* K_n *then :*

1. *Either there exists* $\langle \nu - \kappa \rangle$ *is in* K_0*;*
2. *or there exists* $\langle \nu - \nu{:}p(\vec{x}) \rangle$ *in an* α_i *such that* $0 \leq i < n$*;*
3. *or there exists* $\langle \mu - \nu{:}p(\vec{x}) \rangle$ *in an* α_i *such that* $0 \leq i < n$ *and* $p \in \mathcal{I}_\nu$*,*

Proof. By induction on the length of the derivation. If $n = 0$ we are trivially in case *1*. Otherwise either $\langle \nu - \kappa \rangle$ is in K_{n-1}, in wich case we conclude by the induction hypothesis, or $\langle \nu - \kappa \rangle$ is produced by a *modular CSLD* transition. In the latter case α_{n-1} is of the form $\gamma', \langle \nu' - \nu{:}p(\vec{x}) \rangle, \gamma''$ with $(\nu'{=}\nu) \vee (p \in \mathcal{I}_\nu)$. If $(\nu'{=}\nu)$ we are in case *2*, otherwise we are in case *3*. □

The property of implementation hiding formally states that to enter a module we need to pass through a public predicate of this module.

Proposition 3.6 (Implementation Hiding) *Let* $(c_0|K_0|\alpha_0) \longrightarrow (c_1|K_1|\alpha_1) \longrightarrow \ldots \longrightarrow (c_n|K_n|\alpha_n)$ *be a derivation. If* $\langle \nu - A \rangle$ *belongs to the sequence* α_n *then :*

1. *Either there exists* $\langle \nu - \mu{:}p(\vec{x}) \rangle$ *in* α_0*;*
2. *or there exists* $\langle \nu - \kappa \rangle$ *in* K_0*;*
3. *or there exists* $\langle \mu - \nu{:}p(\vec{x}) \rangle$ *in an* α_i *(*$0 \leq i < n$*) such that* p *is public in the module* ν*;*

Proof. By induction on the length of the derivation. If $n = 0$ then we are trivially in case *1.* Otherwise by lemma 3.4 we know that :

- Either there exists $\langle \nu - \mu{:}p(\vec{x}) \rangle$ in α_{n-1}, in which case the proposition is true by induction hypothesis;
- or there exists $\langle \nu - \kappa \rangle$ in K_{n-1}, in which case the correction is deduced by lemma 3.5 and induction hypothesis;
- or there exists $\langle \mu - \nu{:}p(\vec{x}) \rangle$ in α such that p is public in ν, which corresponds to the case *3.*

□

4 Logical Semantics

4.1 Modules as a Constraint System $\mathcal{M}$

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.

4.2 Translation of MCLP($\mathcal{X}$) Programs into CLP($\mathcal{M}, \mathcal{X}$) Programs

The logical semantics of modular CLP programs is obtained by a formal translation of pure MCLP($\mathcal{X}$) programs into ordinary CLP($\mathcal{M}, \mathcal{X}$) programs. This also 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. The translation Π of the MCLP program is as follows:

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

Table 2: Formal tanslation of MCLP($\mathcal{X}$) to CLP($\mathcal{M},\mathcal{X}$)

Proposition 4.1 (Soundness) *Let $\mathcal{P}$ and $(c|\gamma)$ be a pure MCLP program and a pure MCLP goal*

$$if\ \left((c|\gamma) \xrightarrow{\mathcal{P}} (d|\gamma')\right)\ then\ \left((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d, allow(y,\mu,p), y=\nu|\Pi^{\langle\rangle}(\gamma'))\right)$$

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

Proof. Let us suppose $((c|\gamma) \xrightarrow{\mathcal{P}} (d|\gamma'))$. Let $\left\langle\nu-\mu{:}p(\vec{t})\right\rangle$ be the selected atom in γ. Then γ is of the form $\left(\gamma_1, \left\langle\nu-\mu{:}p(\vec{t})\right\rangle, \gamma_2\right)$ for some γ_1 and γ_2. Hence we have $\Pi^{\langle\rangle}(\gamma) = \left(\Pi^{\langle\rangle}(\gamma_1), \dot{p}(\nu,\mu,\vec{t}), \Pi^{\langle\rangle}(\gamma_2)\right)$. Let $(p(\vec{s}) \leftarrow c'|\alpha)\theta$ be the selected clause in module μ. Then we have $(\dot{p}(y,\mu,\vec{s})) \leftarrow c, allow(y,\mu,p)|\Pi_\mu(\alpha))\,\theta$ in the translation of $\mathcal{P}$. We also have $d = (c, \vec{t}{=}\vec{s}, c')$, $X \models \exists(d)$ 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 $d' = (c, (\nu,\mu,\vec{t}) = (y,\mu,\vec{s}), c', allow(y,\mu,p)$. Therefore we have $((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d'|\Pi^{\langle\rangle}(\gamma')))$. □

Lemma 4.2 *The functions Π_μ and $\Pi^{\langle\rangle}$ are injective.*

Proof. Π_μ on qualified atomes, atom sequences and clauses is a composition of injective functions and therefore is injective. For the same reason $\Pi^{\langle\rangle}$ is injective. Because Π_μ on modules is the pointwise extension of Π_μ defined on clauses, it is injective too.

□

Proposition 4.3 (Completeness) *Let $\mathcal{P}$ and $(c|\gamma)$ be a pure MCLP program and a pure MCLP goal*

$$if\ \left((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d|\alpha)\right) then \left((c|\gamma) \xrightarrow{\mathcal{P}} (d'|\gamma')\right)$$

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

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

The soundness and completeness of the translation thus show that a MCLP($\mathcal{X}$) programs are provided with simple logical semantics given by the ordinary logical semantics of their translations as a CLP($\mathcal{M}, \mathcal{X}$) programs.

5 Pragmatic Aspects

In practice, we deal with a set of *atoms* defined as in ISO Prolog [10] to represent function symbols, predicate symbols and module names. In this chapter, the term atom has thus a different meaning than in the previous sections, and we will use the term *predicate* to refer to the atomic propositions as defined in section 2. As the position of an atom in a clause makes it possible to determine which functor/arity is represented, the two relations $\overset{M}{\rightsquigarrow}$ and $\overset{P}{\rightsquigarrow}$ will be the trivial bijections.

5.1 Defining a Module

A module is a set of clauses and directives contained in a single file named with the name of the module and a `.pl` extension. The file must begin with a `module/2` declaration which specifies the name of the module and its interface. The name of the module is an atom, whereas its interface must be either a list of predicate specifications of the form `Functor/Arity` for representing the set of public predicates, or a '`_`' denoting the fact that all the predicates of the module are public.

5.2 Using a Module

The *use of a module* consists in the import of some predicates and some declarations of the module to the current one. The programmer can use a module by means of the `use_module/2` declaration where the first argument is the name of the module used, and the second argument is the list of imported predicates.

Importing predicates The second argument of the `use_module/2` declaration is a list of the form `[Functor/Arity, ...]` which indicates the set of imported predicates. By using this declaration the programmer states that every imported predicate can be considered as a predicate of the current module. Hence it is possible to declare as public an imported predicate in the `module/2` declaration. In order to guarantee the implementation hiding, only public predicates can be imported. It is forbidden to import a predicate with the same name/arity as a local predicate or a previously imported predicate. If the programmer wants to use more that one predicate with the same name/arity (defined in different modules), at most one must be imported and a complete qualification call of the form `Module:p(Arg_1, ..., Arg_n)` must be used for the other ones. It is also possible to omit the second argument of the `use_module` declaration, in such a case the system assumes that all public predicates of the used module are imported.

Importing Declarations By using the `use_module/2` declaration, some declarations are automatically imported. It is especially the case of syntax declarations. Unlike the import of predicates, the import of declarations are not parameterized, all syntax modifications are imported.

5.3 Visibility

The formal semantics proposed in sections 2 and 3 (or 4) clearly define the visibility of predicates from a particular module. The predicates which can be called from a module are the predicates defined in that module plus all public predicates of imported modules. A call is either a module-qualified predicate (in the form `module:p(X1,..., XN)`) or a non-qualified predicate. If a not fully qualified predicate occurs, the compiler assumes that this call is implicitly made inside the module. In other words, in the module `module` the predicate `p(X_1, ..., X_n)` stands for `module:p(X_1, ..., X_n)`. In the same spirit, the predicate `call(X)` in the module `Module` is a shortcut for `call(Module, X)`.

For other features not modeled in our semantics, such as global variables [8] or attributes [13] clause assertions/retractions etc., we propose a very simple but radical solution : by default only the local work space (work space of the current module) is visible from outside. In other words using such features can only be done inside the module itself. Hence to allow the access to global variables or attributes or dynamic clauses from outside (in order to consult or modify), the programmer must create accessors declared public in the interface.

Nevertheless, in order to simplify the life of programmers who do not want to protect their own modules and do not want to be disturbed by such controls, it is possible to make accessible the work space of any module by declaring it public with the `public/0` directive. For instance, to assert a clause from a module `module1` in a `module2`, one uses a call of the form `module2:assertz(p(T1, ..., TN)`.

5.4 Closures

In order to create and manipulate closures, two operators have been added: `closure/3` and `apply/2`. The predicate `closure(X,G,C)` unifies `C` with the closure made from the goal `G` (supposed to be instantiated at compile-time), in which the variable `X` is abstracted, whereas `apply(C, T)` replaces in the closure `C` the abstraction variable by the term `T` and executes the resulting goal. In practice the system deals with slightly more general closures than those defined in section 2, for instance a goal instead of a predicate can be given as second argument of the `closure/3` operator.

5.5 The Default Module

In our module system we allow the compilation of classical Prolog files without module declaration. The code belonging to such files is considered to belong to the special module `user`. This module is public and access to this module is never controlled by the system. Our system is thus able to execute old GNU-Prolog files. Moreover this module is the default module of every call made from the top level.

5.6 Packages

Despite the fact that the module system we have defined strongly limits conflicts of predicate names, its distributed use leads quickly to a new conflict problem : the conflict of module names. To avoid that, a notion of package similar to the Java language has been introduced. A package is defined as a sequence of atoms separate by slashes. The notion of module name is extended to an atom or a pair package - atom separated by a slash.

For sake of conciseness, the directive `import/1` has been added to import module names. For example, the use of `:-import(oefai/clpr)` indicates to the system that the atom `clpr` refers to the module `clpr` belonging to the package `oefai`. In the same way, with `:-import(oefai/_)`, the user can import all modules of the package `oefai`. The import of two modules with the same name is however forbidden.

5.7 About the Implementation

The actual implementation of our module system consists of a layer, written in Prolog, on the top of GNU Prolog RH [9], a version of GNU Prolog extended with coroutines and attributed variables. This layer is composed of:

- a preprocessor that converts modular code into non-modular code ;
- a library to handle dynamic calls ;
- a new top-level, that transparently interprets modular programs and is able to dynamically compile and load modules and their dependencies.

Attributed variables are used to represent closures, and are useful to protect them from a "reverse engineering" approach, as they forbid the user to hack the internal representation of closures. This is the only reason to use attributed variables in the implementation of the module system.

6 Comparison with other Module Systems

In this subsection, we present a brief overview of existing syntactic module systems and show how they deal with the problem of high-order.

6.1 SICStus Prolog.

In SICStus [27], all predicates are public (formally for all module $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P}$ we have $I_\mu = \Sigma_P$). On the other hand, this system does not provide any kind of code protection. Thanks to the `meta_predicate` directive, that allows to explicitly declare meta-predicates, the system is able to implicitly add the calling context to meta-data. The principal defect of this approach is thus the abandonment of the principle of implementation hiding.

6.2 ECLiPSe.

ECLiPSe [1] proposes an intermediary solution. In fact it provides two different predicates to invoke meta-calls. The first one `:/2` behaves as `call/2` (defined in section 3) whereas the second one `@/2` does not make any permission test. It provides also a directive `tool/1`, (analogous to the `meta_predicate` directive) that allows the implicit addition of the calling context as argument of the meta-predicate. Hence for a common use the programmer employs `:/2` and `@/2` for meta-predicates. This solution has the advantage of limiting the unauthorized calls made in a unconscious way. Nonetheless, as beforehand, it is not possible to ensure the hiding of any predicate. It can be noticed that this proposal is very close to the ISO one [16].

6.3 SWI Prolog.

For meta programming, SWI Prolog [29] uses a slightly different semantics from the one proposed in section 3. In fact, SWI handles two distinct calling contexts. The former, that we call *static context*, plays the role of the calling context in a modular CSLD transition. The latter, that we call *dynamic context*, plays the role of the calling context in a call transition. By default the dynamic context is the same that the static context. However, for a meta-predicate, the dynamic context is the context of the goal that calls it. In SWI, predicates with this behavior are called "module transparent" and are declared with the directive `module_transparent/1`. By declaring `forall/2` as "module transparent", SWI-Prolog executes the previous example as expected.

Nonetheless, a dynamic call does not behave as a static one. Therefore with a "module transparent" predicate, the two goals `p(x)` and `(G=p(x), call(G))` do not call `p/1` in the same module. The former makes the call in the static context, whereas the latter makes it in the dynamic context.

Moreover, these conventions do not provide a code protection as strong as we want. For instance, the use of the implementation of the `findall/3` predicate (defined in the example 2.6) does not work better either in SWI, all assertion/retraction being made in the calling module instead of the called module.

6.4 CIAO Prolog.

Conceptually, CIAO Prolog [3] adopts a closed module system. In order to allow the manipulation of meta-data through the module system, it provides an advanced version of the directive `meta_predicate/1`. Before calling the meta-predicates, the system compiles on the fly meta-data following a method analoguous to the one presented in section 4. Since this compilation is done before the call of the meta-predicate, the system is knows the calling context in which the meta-data must be called, to obtain the expected result. The meta-predicate handles then a compiled version of the meta-data that is possible to invoke with the predicate `call/1`. As far as the system does not document any predicate (except `call/1`) able to create or manipulate such compiled meta-data, the implementation hiding property is preserved.

Although not clearly stated in the manual, CIAO Prolog does make a distinction between terms and higher-order data (i.e. compiled versions of goal). However, instead of hiding these objects behind a directive, we propose to manipulate closures as first-class citizens.

6.5 XSB

The XSB system [23] is also considered as a syntactic system, but follows a quite different approach. In fact, this system belongs to the class of *atom-based* systems, rather than *predicate based*. The main difference is that XSB modularizes function symbols too. Hence two compound terms constructed in two different modules can not be unified. In a module, it is possible however to import public symbols from another module, the system considers then that the two modules share the same symbols. The semantics of the `call/1` predicate is hence 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.

However, this solutionmainly moves the problem to the construction of the terms. Indeed in XSB, the terms constructed with `=../2`, `functor/2` and `read/1` are supposed to belong to the module `user`. As a consequence, the system does not respect anymore the independence of the selection strategy. For instance, in a module different form `user`, the goal `(functor(X,foo,1), X=foo(_))` fails, whereas `(X=foo(_), functor(X,foo,1))` succeeds.

7 Conclusion

In a classical 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 logic programs with modules, and that the protection of private module predicates necessitates to distinguish between term calls (meta-programming) and closures (higher-order).

The module system we propose is close to the one of CIAO Prolog in its implementation, but has the advantage of revealing the need for closures, and of positioning them w.r.t. meta-programming predicates in the framework of formal operational semantics. Furthermore, an equivalent logical semantics has been provided for pure modular programs.

Our module system has been implemented in GNU-Prolog. Some existing code has been ported as libraries using the module system, such as for instance an implementation of CHR [26]. This modularization of CHR provides an example of intensive use of the module system allowing the development of several layers of constraint solvers in CHR. Other libraries have been developed with this closed module system for the development of SiLCC. One can quote a dynamic lexer-parser allowing modular redefinitions of the syntax with a powerful generalization of the directive `op/3`. This library belongs to the bootstrap libraries of our on-going implementation of SiLCC.

As for future work, we can mention the possible implementation of so-called *functors* [18] allowing the static instantiation of parameterized modules, and the investigation of object-oriented programming features with modules and global variables in CLP.

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 to Daniel de Rauglaudre and Olivier Bouissou for their comments and help in the implementation.

References

[1] Abderrahamane Aggoun and al. *ECLiPSe User Manual Release 5.2*, 1993 – 2001.

[2] 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.

[3] F. Bueno, D. Cabeza 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.

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

[5] Daniel Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid, August 2004.

[6] Daniel Cabeza and Manuel 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.

[7] Weidong Chen. A theory of modules based on second-order logic. In *The fourth IEEE. Internatal Symposium on Logic Programming*, pages 24–33, 1987.

[8] Daniel Diaz. *GNU Prolog user's manual*, 1999–2003.

[9] Daniel Diaz and Rémy Haemmerlé. *GNU Prolog RH user's manual*, 1999–2004.

[10] Pierre Deransart Adbelali. Ed-Dbali and Laurent. Cervoni. *Prolog: The Standard.* Springer-Verlag, New York, 1996.

[11] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, February 2001.

[12] Patrica M. Hill. A parameterised module system for constructing typed logic programs. Technical Report 93.12, University of Leeds, Mar 1993.

[13] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. Rapport Technique TR-92-23, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1992.

[14] C. Holzbaur. Oefai clp(q,r) manual rev. 1.3.2. Technical Report TR-95-09, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1995.

[15] International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 1: General core*, 1995. ISO/IEC 13211-1.

[16] International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 2: Modules*, 2000. ISO/IEC 13211-2.

[17] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[18] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

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

[20] Luís Monterio and António Porto. Contextual logic programming. In *Proceedings of ICLP'1989,International Conference on Logic Programming*, pages 284–299, 1989.

[21] Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language.* PhD thesis, Department of Informatics, University of Beira Interior, Portugal, September 2003.

[22] Richard A. O'Keefe. Towards an algebra for constructing logic programs. In *Symposium on Logic Programming*, pages 152–160. IEEE, 1985.

[23] Konstantinos Sagonas and al. *The XSB System Version 2.5 - Volume 1: Programmer's Manual*, 1993 – 2003.

[24] D. T. Sannella and L. A. Wallen. a calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, pages 147–177, 1992.

[25] Vijay A. Saraswat and Patrick Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

[26] Tom Schrijvers and David 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.

[27] 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.

[28] David 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] Jan Wielemaker. *SWI Prolog 5.4.1 Reference Manual*, 1990– 2004.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399