Un Système de Module Fermé pour la PLC

Rémy Haemmerlé, Francois Fages

Un Système de Modules Fermé pour la PLC

Rémy Haemmerlé **François Fages**

INRIA, Projet Contraintes
Domaine de Voluceau, Rocquencourt
BP 105, 78153 Le Chesnay Cedex, France

Remy.Haemmerle@inria.fr Francois.Fages@inria.fr

Résumé

Cet article présente un système de modules pour la Programmation Logique par Contraintes. Ce système simple et relativement indépendant du langage précis d'utilisation (il a été conçu à l'origine pour GNU-Prolog) a la particularité d'être *fermé*. Par fermé nous entendons, le fait qu'il est capable de prévenir tout appel à des parties privées d'un module depuis l'extérieur de celui-ci. Bien qu'il puisse être vu comme une simple couche de sucre syntaxique, ce système offre une discipline de nommage des prédicats permettant de développer des bibliothèques et de faciliter la réutilisation du code. De plus, en ajoutant au langage une forme de fermeture inspirée de la programmation concurrente linéaire par contraintes, nous montrons comment on résout le problème bien connu de l'utilisation de la méta-programmation à travers le système de modules, en distinguant l'exécution d'un terme (prédicat `call`) de l'application d'une fermeture (ordre supérieur).

Abstract

This article presents a module system for the Logic Constraint Programming. This system simple and pretty independent a precise language (it was created at the origin for GNU-Prolog) has the particularity to be *closed* . By closed we meen the fact that the system is able to prevent any call to private parts of a module from the exterior of this one. Although it can be seen like a simple layer of syntactic sugar, this system offers a discipline of predicates naming, making possible the developpement of libraries and facilitating the re-use of the code. Morevocer, by adding a form of closures inspired by Linear Concurrent Constraint programming to the language, we show how the well-known problem of using meta-programming through the module system can be solved by distinguishing the execution of a term (predicate `call`) from the sending of a closure (higher order).

1 Introduction

Il est souvent plus facile de travailler avec un nombre restreint de concepts à l'esprit. Ainsi pour être écrit proprement, les programmes de grande taille doivent être divisés en bloc de taille plus limitée. Bien que cela puisse être fait dans n'importe quel langage, un système de modules approprié facilite la segmentation des programmes en obligeant le programmeur à définir clairement la limite et l'interface de chacun de ces blocs. Beaucoup d'erreurs peuvent être alors évitées, le compilateur étant en charge de vérifier que les appels entre modules respectent les interfaces déclarées. Une telle segmentation du code favorise aussi la compréhension d'un programme par des personnes extérieures et améliorent ainsi sa réutilisation.

Un des objectifs de notre équipe[1] est de concevoir une implémentation de la Programmation Linéaire Concurrente par Contraintes (LCC) [12] basée sur la technologie de compilation en code natif de GNU-Prolog [8]. Avant de commencer à proprement parlé le développement de ce système, que nous avons appelé SiLCC pour «SiLCC is Linear Concurrent Constraint programming», il a été nécessaire de concevoir un système de modules[2]. L'objectif de cet article est de présenter les choix théoriques et pratiques que nous avons faits. Pour des raisons de simplicité et parce que nous pensons, comme Leroy [18] que la programmation modulaire «a peu à voir avec les particularités de n'importe quel langage de programmation» nous présenterons notre système de modules dans le contexte de la Programmation Logique par Contraintes (PLC) [17] plutôt que dans le contexte plus général de LCC.

[1] Projet Contraintes, INRIA `http://contraintes.inria.fr`
[2] puisque GNU-Prolog n'a pas de système de modules.

1.1 Objectifs

Nous commençons en présentant ici les objectifs principaux qu'un système de modules doit remplir :

Protection du Code. Un programmeur doit pouvoir protéger son code contre des intrusions faites par du code étranger. Cela signifie qu'il doit être possible de limiter l'accès à un module depuis l'extérieur de celui-ci. Par la suite, nous dirons qu'un système de modules qui assure une protection de code est un système *fermé*, dans le cas contraire nous le qualifierons d'*ouvert*.

Séparation de l'espace des noms. Dans les systèmes sans module, les conflits de noms sont fréquents et perturbants. En effet, pour être sûr d'éviter de tels problèmes, le programmeur est souvent contraint de préfixer systématiquement le nom de tous ses prédicats. Un système de modules permet d'éviter de façon transparente les conflits de noms.

Compilation séparée. Il doit être possible de compiler indépendamment chaque module d'un programme. En particulier les bibliothèques standards seront compilées une fois pour toutes.

Boostrap de SiLCC. Un trait particulièrement original de SiLCC est la réalisation d'un langage de programmation par contraintes entièrement bootstrappé à partir d'un petit noyau (LCC avec contraintes de graphes étiquetés). Le système de modules est fondamental au bootstrap.

Par ailleurs le système de modules doit satisfaire d'autres critères :

Simplicité. Nous ne voulons pas que le système de modules limite les facilités de Prolog pour le développement rapide. En particulier :
- La concision des programmes Prolog doit être préservée, nous souhaitons éviter notamment d'introduire trop de déclarations obligatoires.
- Le système doit permettre la méta-programmation (c.à.d. fournir le prédicat `call`), qui nous semble particulièrement utile pour écrire un top-level ou des meta-interpréteurs.
- Nous voulons éviter l'introduction de trop de nouveaux concepts qui pourraient perturber les habitudes des programmeurs.

Considérations sémantiques. Notre système ainsi que ses propriétés doivent avoir une définition formelle.

Implémentation facile. Le système de modules doit être facile à coder et à porter au-dessus de différents langages logiques. En effet puisque la conception de SiLCC n'est pas terminée, nous voulons éviter la réécriture complète du système chaque fois qu'un nouveau trait sera ajouté au prototype.

1.2 Travaux antérieurs

La modularité dans le contexte de la programmation logique a été considérablement étudiée. Afin de rendre compte des notions importantes de modularité, telles que le masquage d'informations ou les relations d'import/export, certaines approches utilisent des extensions de la programmation logique classique. Nous pouvons citer par exemple les extensions par des implications imbriquées [19], en méta logique [3] ou en logique du second ordre [7]. D'autres approches, comme la programmation logique contextuelle [20] ou l'extension orientée objet [21] vont plus loin en préconisant l'adoption de concepts plus puissants qu'un système de modules statiques.

Plusieurs autres propositions ajoutent des constructions permettant de déclarer et de manipuler des modules. D'un coté il y a les calculs de modules de O'Keefe [22] ou de Sannella et Wallen [24] inspirés de la programmation fonctionnelle. De l'autre il y a les systèmes dits syntaxiques (parce qu'ils ne s'occupent que de la table des symboles) utilisés par la plupart des systèmes Prolog courants (par exemple SICStus [26], SWI [28], ECLiPSe [1], XSB [23], Ciao [4, 5]). Néanmoins aucun d'entre eux ne respecte tous les objectifs que nous avons présentés précédemment. Par exemple, le système SICStus ne fournit aucune protection du code.

Bien que les systèmes syntaxiques aient été critiqués (par exemple dans [22, 24]), nous avons décidé, comme Cabeza et Hermenegildo [6], d'adopter ce choix. Les raisons principales en sont sa simplicité et la compatibilité offerte avec les bibliothèques existantes comme CLP(q, r) de Holzbaur [14] ou CHR de Schrijvers [25], que nous projetons d'employer.

1.3 Plan

Le papier est organisé comme suit. La section 2 propose une sémantique formelle pour les systèmes syntaxiques dans le contexte formel de la PLC avec méta-programmation. Nous montrons également que ce système garantit la protection du code. Dans la section 3 nous donnons un méthode formelle pour traduire des programmes modulaires contenant des méta-appels en programmes PLC purs (comme définis dans [17]). Puis, à la section 4, nous présentons l'utilisation pragmatique de notre système. La section 5 présente le problème que pose l'utilisation de l'ordre supérieur à travers un système de modules, et les difficultés que cela posent dans les systèmes syntactiques existants. Finalement nous introduisons une forme de

fermeture, inspirée de LCC, qui permet de résoudre ces problèmes.

2 PLC Modulaire

Nous présentons ici une sémantique formelle pour la Programmation Logique par Contraintes Modulaire (PLCM), et son extension pour la métaprogrammation.

2.1 Syntaxe et définitions

Dans la suite on suppose que :
- V est un ensemble de variables dénombrable ;
- Σ_F est l'ensemble des symboles de fonctions ;
- Σ_C est un ensemble de symboles de prédicats (l'ensemble des contraintes) ;
- Σ_P est l'ensemble des symboles de prédicats (disjoint de S_C) ;
- Σ_M est l'ensemble des identifiants de modules.

Pour des raisons de simplicité, nous supposons ici que tout atome est qualifié and ne décrivons pas les conventions standard utilisées pour préfixer automatiquement les atomes donnés sans identifiant de module.

Définition 2.1 (Clause) *Une* clause PLCM *est une formule*

$$A_0 \leftarrow c_1, \ldots, c_n | \mu_1 : A_1, \ldots \mu_m : A_m.$$

où les c_i sont des contraintes atomiques, les A_i sont des atomes et les μ_i's sont des identifiants de modules. Dans la suite, nous appellerons atomes qualifiés *les $(\mu_i : A_i)$ et une clause sera notée de façon concise par $A \leftarrow c|\alpha$.*

Définition 2.2 (Module) *Un* module *est un n-uplet $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)$ où $\mu \in \Sigma_M$ est le* nom *du module, $\mathcal{D}_\mu$ un ensemble de clauses est appelé* implémentation *du module et $\mathcal{I}_\mu \subset \Sigma_P$ est l'*interface *du module. Si un symbole p appartient à l'interface d'un module μ, nous dirons que p est* public *dans μ.*

Définition 2.3 (Programme) *Un* programme PLCM *$\mathcal{P}$ est un ensemble de modules dont les noms sont distincts.*

Définition 2.4 (But) *Un* but PLCM *est une formule*

$$c_1, \ldots, c_n | \langle \nu_1 - \mu_1 : A_1 \rangle, \ldots, \langle \nu_m - \mu_m : A_n \rangle$$

où les c_i sont des contraintes atomiques, les $(\mu_j : A_j)$ sont des atomes qualifiés et les ν_j sont des identifiants de modules appelés dans ce cas contexte d'appel. *$\langle \nu_j - \mu_j : A_j \rangle$ sera appelé* atome avec contexte.

Dans la suite $\langle \nu - (\mu_1 : A_1, \ldots, \mu_n : A_n) \rangle$ sera une notation simplifiée pour représenter la séquence d'atomes avec contexte $(\langle \nu - \mu_1 : A_1 \rangle, \ldots, \langle \nu - \mu_n : A_n \rangle)$.

Example 1. Pour illustrer le problème de l'utilisation du `call`, étudions la définition classique du predicat `findall/3` :

```
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).
```

L'utilisation du `call` dans ces déclarations, n'empêche pas l'intrusion d'un autre module par un appel comme `findall(X,retract(found(X)),L)`. Pour ce prémunir de ce genre de comportement, nous proposons la sémantique qui suit

2.2 Sémantique Opérationnelle

Définition 2.5 (Transition) *Soit $\mathcal{P}$ un programme PLCM. La relation de réécriture $\longrightarrow$ sur les buts est définie par la plus petite relation satisfaisant le principe* résolution CSLD modulaire *suivant.*

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

Ici le principe de résolution CSLD classique [11] est restreint par la condition $(\nu = \mu) \vee (p \in \mathcal{I}_\mu)$. Celle ci traduit le fait que $\mu : p(\vec{t})$ ne peut être exécuté que si l'appel est fait à partir du contexte μ ou que si le prédicat p est public dans le module μ.

Lemme 2.6 *Si $(c|\gamma) \longrightarrow (d|\gamma', \langle \nu - A \rangle, \gamma'')$ est une transition alors il existe un atome avec contexte $\langle \mu - \mu' : p(\vec{t}) \rangle$ dans la séquence γ tel que :*
- *soit $\nu = \mu = \mu'$,*
- *soit $\nu = \mu'$ et p est public dans ν.*

Proposition 2.7 (Masquage de l'implémentation) *Soit $(c_0|\gamma_0) \longrightarrow \ldots \longrightarrow (c_n|\gamma_n)$ une dérivation. Si l'atome $\langle \mu - A \rangle$ appartient à la séquence γ_n alors :*
- *Soit il existe un atome $\langle \mu - B \rangle$ dans γ_0,*
- *Soit il existe un atome $\langle \nu - \mu : p(\vec{t}) \rangle$ dans une séquence γ_i telle que $0 \leq i \leq n$ et p soit public dans ν.*

Nous verrons à la fin de la section suivante que $\longrightarrow$ préserve l'indépendance de la stratégie de sélection des atomes [11].

2.3 Méta-Programmation

On suppose que $call/2 \notin \Sigma_P$ et que $\overset{M}{\rightsquigarrow}: \Sigma_F \times \Sigma_M$ et $\overset{P}{\rightsquigarrow}: \Sigma_F \times (\Sigma_P \cup \{call/2\})$ sont 2 relations appelées "de conversion". Comme Σ_F, Σ_P et Σ_M sont supposés être distincts les deux relations $\overset{P}{\rightsquigarrow}$ et $\overset{M}{\rightsquigarrow}$ seront utiles pour interpréter les symboles de fonction comme des symboles de prédicat et des identifiants de module lors d'un méta-appel. Nous étendons maintenant la notion de programmes PLCM, en autorisant l'utilisation de $call/2$ comme symbole de prédicat dans un but et dans le coté droit d'un clause. Nous notons la nouvelle classe de langage obtenue PLCM$^+$.

Définition 2.8 (Méta-Transition) *Soit $\mathcal{P}$ un programme PLCM$^+$. La relation de réécriture $\longrightarrow^+$ sur les buts est définie comme la plus petite relation contenant $\longrightarrow$ et satisfaisant la règle suivante :*

$$\frac{\mathcal{S} \models \exists(c \wedge t{=}g \wedge s{=}f(\vec{x})) \qquad f \overset{P}{\rightsquigarrow} p \qquad g \overset{M}{\rightsquigarrow} \mu}{\begin{array}{c}(c|\gamma, \langle \nu - \nu{:}call(t,s)\rangle, \gamma') \longrightarrow^+ \\ (c, t{=}g, s{=}f(\vec{x})|\gamma, \langle \nu - \mu{:}p(\vec{x})\rangle, \gamma')\end{array}}$$

Faisons quelques remarques :
- $\longrightarrow^+$ préserve l'indépendance de la stratégie de sélection.
- Comme un méta-appel ne change pas le contexte d'appel, $\longrightarrow^+$ préserve aussi le masquage de l'implémentation.
- Par souci de simplicité, la définition formelle de $call/2$ n'autorise pas le méta-appel d'une conjonction d'atomes ni le méta-appel d'une contrainte. Cependant, afin d'émuler les méta-appels de conjonctions, on pourrait supposer $(',')/2 \overset{P}{\rightsquigarrow} and/2)$ et ajouter à l'implémentation de tout module μ du programme la clause $(and(x,y) \leftarrow \mu : call(x), \mu : call(y))$. Les méta-appels de contraintes peuvent se traiter de manière analogue.
- Suivant la sémantique donnée, le but $(c|\langle \mu - \nu{:}call(t,s)\rangle)$ réussit même si les arguments du *call* sont des variables libres. Ce cas traduit une erreur du programmeur et soulever une exception est le plus approprié. Cependant notre sémantique ne prenant pas en compte la notion d'exception, le choix du *succès* sans instanciation, plutôt que l'*échec*, est préférable car il permet de préserver l'indépendance de la stratégie de sélection. Cela peut être illustré par le but suivant : `(X=true, call(X))`. En supposant que le méta-appel d'une variable libre échoue, l'emploi d'une stratégie de sélection gauche-à-droite mène à la réponse calculée `X=true` tandis qu'une stratégie droite-à-gauche ne mène à aucune réponse.

3 Compilation Formelle

Nous avons l'intention d'exécuter les programmes modulaires sur une machine abstraite proche de celle de Warren [2]. Pour montrer que cela est réalisable, nous proposons ici une traduction formelle des programmes de PLCM(S) vers des programmes PLC(S) pures.

3.1 Compilation des Programmes PLCM

Dans cette section, nous supposons que $\pi : (\Sigma_M \times \Sigma_P \times \mathbb{B}) \rightarrow \Sigma'_P$ est une une fonction injective telle que Σ'_P soit disjoint de Σ_C. Ainsi à chaque couple (μ, p) nous serons capable de faire correspondre 2 symboles de prédicats appartenant à Σ'_P. Le premier, $q' = \pi(\mu, p, \mathbf{0})$, représentera la version privée de p dans μ, alors que le deuxième, $q'' = \pi(\mu, p, \mathbf{1})$, représentera la version publique de p dans μ. Nous définissons ci-dessous, une fonction de traduction Π des programmes PLCM(S) vers les programmes PLC(S).

$$\Pi^{:}_{\mu}(\nu{:}p(\vec{t})) = (\pi(\mu, p, (\mu \neq \nu \vee p \in \mathcal{I}_\mu)))\,(\vec{t}).$$

$$\Pi^{'}_{\mu}(A_1, \ldots, A_n) = \Pi^{:}_{\mu}(A_1), \ldots, \Pi^{:}_{\mu}(A_n)$$

$$\Pi^{\leftarrow}_{\mu}(p(\vec{t}) \leftarrow c|\alpha) = (\pi(\mu, p, (p \in \mathcal{I}_\mu))(\vec{t})) \leftarrow c|\Pi^{'}_{\mu}(\alpha)$$

$$\Pi_\mu\left(\bigcup\{(A \leftarrow c|\alpha)\}\right) = \bigcup\left\{\Pi^{\leftarrow}_{\mu}(A \leftarrow c|\alpha)\right\}$$

$$\Pi\left(\bigcup\{(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)\}\right) = \bigcup\{\Pi_\mu(\mathcal{D}_\mu)\}$$

$$\Pi^{\langle\rangle}\left(\langle \mu - A\rangle\right) = \Pi^{'}_{\mu}(A)$$
$$\Pi^{\langle\rangle}\left((\gamma, \gamma')\right) = \Pi^{\langle\rangle}(\gamma), \Pi^{\langle\rangle}(\gamma')$$

Proposition 3.1 (Correction) *Soit $\mathcal{P}$ et $(c|\gamma)$ un programme et un but PLCM.*

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

Lemme 3.2 *La fonction de traduction Π est injective.*

Proposition 3.3 (Complétude) *Soit $\mathcal{P}$ et $(c|\gamma)$ un programme et un but PLCM.*

$$\begin{array}{c} si\ \left((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d|\alpha)\right)\ alors \\ \exists\gamma'.\left(\Pi^{\langle\rangle}(\gamma') = \alpha \wedge \left((c|\gamma) \xrightarrow{\mathcal{P}} (d|\gamma')\right)\right)\end{array}$$

Corollaire 3.4 $\longrightarrow$ *sur les programmes PLCM préserve l'indépendance par rapport à la stratégie de sélection des atomes.*

3.2 Compilation des programmes PLCM$^+$

Nous définissons maintenant une fonction de traduction Π^+ des programmes PLCM$(\mathcal{S})^+$ vers les programmes PLC$(\mathcal{S})$ purs.

$$\Pi^+_\mu(D_\mu) = \Pi_\mu(D_\mu) \cup \left\{ \begin{array}{c} (\pi(\mu, call/2, \mathbf{0}))\,(g, f(\vec{x})) \leftarrow \Pi^:_\mu(\nu, p(\vec{x})) \\ \text{such as } \left(f \overset{P}{\rightsquigarrow} p \wedge g \overset{M}{\rightsquigarrow} \mu\right) \end{array} \right\}$$

$$\Pi^+\left(\bigcup\{(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)\}\right) = \bigcup\left\{\Pi^+_\mu(D_\mu)\right\}$$

3.3 Remarques

- La méthode de compilation présentée précédemment permet une compilation totalement séparée. En d'autres termes, il est possible de compiler un module sans avoir à compiler (ou préprocesser) le code d'un autre module. En effet la fonction Π_μ est définie sans utiliser la moindre donnée extérieure au module μ
- La compilation des programmes PLCM pures produit du code efficace. En effet la clause produite par la traduction Π est plus simple que son antécédente : chaque couple (symbole de prédicat – identifiant de module) d'un atome qualifié est remplacé par un unique symbole de fonction de Σ'_P. On peut aussi remarquer qu'il n'est pas nécessaire de se rappeler du contexte d'appel, évitant ainsi tout test dynamique de permission.
- D'un point de vue pratique la propriété du lemme 3.2 est intéressante dans le contexte de l'analyse de bogues. Puisque Π est une injection, nous pouvons calculer son inverse et par conséquent, sans ajouter la moindre information de compilation, il est possible, de retrouver la position précise dans le programme original où une erreur a été détectée pendant la compilation (ou l'exécution) de sa traduction.

4 Le Système de Modules d'un point de vue Pragmatique

En pratique on dispose d'un ensemble d'atomes comme défini dans ISO Prolog [10] . Comme dans tout système Prolog nous utilisons ces atomes pour représenter les symboles de prédicat et de fonction ainsi que les identifiants de module, la position des atomes dans les clauses permettant de déterminer à quel couple foncteur/arité ils correspondent. Ainsi les deux relations de conversion $\overset{M}{\rightsquigarrow}$ et $\overset{P}{\rightsquigarrow}$ seront les bijections triviales.

4.1 Création de Modules

Un module est un ensemble de clauses et de directives contenues dans un unique fichier, dont le nom est identique à l'identifiant du module, suivi de l'extension '.pl'. Ce fichier doit commencer par la directive `module/2` indiquant le nom du module et son interface. Le nom du module est un atome tandis que son interface doit être une liste de spécification de prédicat sous la forme `[Foncteur/Arité|...]` représentant l'ensemble des prédicats publics du module. L'interface peut être remplacée par `_`, ce qui impose que tous les prédicats du module soient publics.

4.2 Utilisation de Modules

Un module peut être utilisé par un autre module, en important un certains nombre de prédicats publics et de directives. Le programmeur peut utiliser un module au moyen de la directive `use_module/2` où, le premier argument doit être le nom du module utilisé.

4.2.1 Import de Prédicats

Le deuxième argument de la directive `m:use_module` est une liste de la forme `[Foncteur/Arité|...]` indiquant quels prédicats doivent être importés. Par le biais de cette directive le programmeur stipule que tout prédicat importé doit être considéré comme un prédicat du module courant. Par conséquent il est possible de déclarer comme public (à l'aide de la directive `module/2`) un prédicat importé. Afin de garantir le masquage de l'implémentation d'un module utilisé, seuls les prédicats publics peuvent être importés. De plus, on interdit l'import de prédicats ayant le même foncteur et la même arité qu'un prédicat local ou qu'un prédicat précédemment importé.

Si le programmeur veut utiliser plusieurs prédicats ayant le même nom et la même arité et définis dans différents modules, il doit vérifier au préalable qu'au plus un est importé et utiliser un appel correctement qualifié de la forme `Module:p(Arg_1, ..., Arg_n)` pour les autres. Il est également possible d'omettre le deuxième argument de la directive `use_module`, au quel cas le système suppose que tous les prédicats du module utilisé doivent être importés.

4.2.2 Import de Directives

L'utilisation de la directive `use_module/2`, impose que certaines directives soient importées. C'est notamment le cas des déclarations de syntaxes. On peut noter que, au contraire de l'import de prédicats, l'import de directives ne peut être paramétré.

4.3 Visibilité

En ce qui concerne les appels classiques et les méta-appels, les sémantiques proposées dans la section 2 définissent clairement les règles de visibilité de prédicats. Les seuls prédicats qui peuvent être appelés à partir du module sont les prédicats définis dans le module lui même, plus tous les prédicats publics du programme. Un appel peut être soit un prédicat qualifié de la forme `module:p(X_1..., X_n)` soit un prédicat non qualifié, le compilateur supposant alors que cet appel est implicitement fait à l'intérieur du module courant. En d'autres termes, dans le module `Module` le prédicat `p(X_1..., X_n)` représente `Module:p(X_1..., X_n)`. Dans le même esprit, le prédicat `call(X)` est vue comme un raccourci pour le prédicat `call(Module, X)`.

Pour d'autres traits non modélisés dans notre sémantique (telles que les accès aux variables globales [9], la manipulation d'attributs [13], les assertions dynamiques de clauses etc..), nous proposons une solution simple mais radicale : par défaut seul l'espace de travail local (l'espace de travail du module courant) est visible. En d'autres mots, l'emploi de telles fonctionnalités ne peut être fait qu'à l'intérieur même d'un module. Par conséquent pour permettre l'accès (afin de consulter ou modifier) aux variables globales, aux attributs ou aux clauses dynamiques depuis l'extérieur d'un module, le programmeur doit créer des prédicats accesseurs déclarés publics.

Néanmoins, afin de faciliter la programmation, on autorise de rendre accessible l'espace de travail de n'importe quel module en le déclarant *public* (au moyen de la directive `public/0`). Le programmeur utilisera la qualification afin d'accéder à l'espace de travail d'un module particulier. Par exemple pour ajouter dynamiquement une clause dans un module public `m1` à partir d'un module `m2`, il emploiera un appel sous la forme `m1:assertz(p(T_1..., T_n)`.

4.4 Module par Défaut

Notre système de modules contient un module par défaut qui permet la compilation de fichiers Prolog classiques (c.à.d. sans déclarations relatives aux modules). Le code de tels fichiers est considéré comme appartenant au module spécial `user`. Ce module est considéré comme public, c.à.d. qu'aucun accès à ce module n'est contrôlé par le système. Ainsi le système peut exécuter des programmes ISO-Prolog. Ce module est aussi le contexte par défaut de chaque appel fait à partir du top-level.

4.5 Les Paquetages

Bien que le système de modules que nous avons défini jusqu'à maintenant limite grandement les conflits de noms de prédicat, son utilisation distribuée aboutit rapidement à de nouveaux problèmes de conflits dérangeant eux aussi : les conflits de nom de modules. Pour éviter cela, une notion de *paquetage* similaire au langage Java est introduite. Un paquetage est défini comme une séquence d'atomes séparés par des slashs. La notion d'identifiants de module est étendue à soit des atomes, soit des paires (paquetage, atome) séparés par un slash. Pour des raisons de concision, la directive `import/1` a été ajoutée au système. Grace à cette dernière il est possible d'importer des noms de module. Par exemple, l'utilisation de la la directive `:-import(Oefai/clpr)` indique au système que l'atome `clpr` fait référence au module `clpr` du paquetage `Oefai`. De même, par l'intermédiaire du paterne `:-import(Oefai/_)` l'utilisateur pourra importer tous les noms de module contenus dans le paquetage `Oefai`. Pour éviter tout problème d'identification, l'import de deux modules de même nom est interdit par le système.

4.6 A propos de l'Implémentation

L'implémentation de notre prototype de système de modules, consiste en une couche ajoutée au dessus de GNU Prolog RH [9], version de GNU Prolog étendue par la gestion des coroutines et des variables attribuées.

Cette couche, écrite en Prolog, est composée :

- d'un préprocesseur qui convertit le code modulaire en code non mondulaire ;
- d'une bibliothèque pour la gestion des appels dynamiques ;
- d'un nouveau top-level, qui interprète de façon transparente les programmes modulaires et qui est capable de compiler et charger dynamiquement des modules et leurs dépendances.

5 Le Problème de l'Ordre Supérieur

Les principes de visibilité des prédicats dans les modules empêchent la manipulation des prédicats en tant que données d'un module à un autre, et interdisent donc l'utilisation de la méta-programmation à travers le système de modules. Suivant la terminologie de la littérature, appelons *méta-prédicats* les prédicats qui manipulent des *méta-données*, c'est-à-dire des données d'ordre supérieur évaluables comme des prédicats.

5.1 Exemple

Soit le predicat `forall/2` défini dans le module `lists` et vérifiant que tous les éléments d'une liste respectent une certaine propriété passée en argument.

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

Maintenant prenons un predicat `toto/1`, supposé être privé dans un autre module. Dans ces conditions l'appel `forall([1,2,3],toto)` échouera systématiquement et ce même si le but `G` est correctement qualifié.

5.2 Solutions Existantes

Dans cette sous-section, nous proposons un bref aperçu des systèmes de modules syntaxiques existant et présentons comment ils traitent le problème de l'ordre supérieur. Nous expliquons, à chaque fois pourquoi nous n'avons pas choisi les solutions qu'ils préconisent.

5.2.1 Solution Naïve

Pour contourner le problème présenté précédemment, le programmeur peut déclarer comme public tout prédicat qu'il emploie comme méta-donnée. D'autre part le méta-prédicat a besoin d'un argument supplémentaire pour permettre au programmeur d'indiquer dans quel contexte l'appel à la méta-donnée doit être fait. Cette méthode réduit néanmoins la protection du code, le système n'étant plus capable de bloquer les appels aux prédicats utilisés en tant que méta-donnée.

5.2.2 SICStus Prolog

Dans SICStus [26] tout prédicat est public (formellement $\forall(\mu,\mathcal{D}_\mu,\mathcal{I}_\mu)\in\mathcal{P}\ \ I_\mu=\Sigma_P$). Ainsi le programmeur peut employer la solution naïve proposée ci-dessus sans avoir a déclarer de prédicat public ; en contrepartie, ce système ne fournit aucune sorte de protection du code. Grâce à la directive `meta_predicate`, qui permet de déclarer explicitement les méta-prédicats, le système est capable d'ajouter implicitement le contexte d'appel aux méta-données. Le principal défaut de cette approche est cependant l'abandon de toute forme de protection des accès aux prédicats.

5.2.3 ECLiPSe

ECLiPSe [1] propose une solution intermédiaire. En effet il fournit deux prédicats différents pour invoquer des méta-appels. Le premier `:/2` se comporte comme `call/2` (défini à la section 2.3) alors que le deuxième `@/2` ne fait aucun test de permission. Il fournit, aussi, une directive `tool/1`, (analogue à la directive `meta_predicate` de SICStus) qui permet d'ajouter implicitement le contexte d'appel comme argument supplémentaire du méta-prédicat. Ainsi pour une utilisation courante, le programmeur emploie `:/2`, tandis qu'il utilise `@/2` quand il conçoit des méta-prédicats. Cette solution a l'avantage de limiter les appels non autorisés, faits de manière inconsciente. Néanmoins, comme dans le cas précédent, il est impossible d'assurer le masquage d'un prédicat. Il est à noter que la solution employée par ECLiPSe est très proche de la proposition ISO [16].

5.2.4 SWI Prolog

Pour la méta-programmation SWI-Prolog [28], utilise une sémantique quelque peu différente de celle que nous avons proposée à la section 2. En fait SWI manipule deux contextes d'appel distincts. Le premier, que nous appelons *contexte statique* joue le rôle de contexte d'appel dans la règle définie à la la section 2.2. Le deuxième que nous appelons *contexte dynamique* joue le rôle de contexte d'appel dans la règle de méta-transition définie à la section 2.3. Par défaut le contexte dynamique est le même que le contexte statique. Pour un méta-prédicat cependant, le contexte dynamique est le contexte dynamique du but qui l'a appelé. Dans SWI-Prolog, les prédicats ayant un tel comportement sont appelés «*module transparent*» et sont déclarés à l'aide de la directive `module_transparent/1`. Avec un tel comportement, et en déclarant `forall/2` comme un prédicat «module transparent», SWI-Prolog exécute l'exemple précédant comme on le souhaite.

Néanmoins un appel dynamique ne se comporte pas comme un appel statique. Par exemple, dans un prédicat «module transparent», les deux buts `p(X)` et `(G=p(X), call(G))` n'appellent pas `p/1` dans le même module (le premier fait l'appel dans le contexte statique tandis que le deuxième le fait dans le contexte dynamique). De plus, il ne fournit pas une protection du code aussi importante que nous le souhaitons. Par exemple, la définition de `findall/2` de la section 2.1 ne fonctionne pas mieux en SWI, toutes les assertions/rétractions étant faites cette fois ci dans le module appelant au lieu du module appelé.

5.2.5 CIAO Prolog

Conceptuellement, CIAO Prolog [4] adopte, un système de modules fermé. Pour permettre la manipulation de méta-données à travers le système de mod-

ules, le système en question fournit une version évoluée de la directive `meta-predicate/1`. Avant l'appel aux méta-prédicats, le système compile à la volée, la méta-donnée en utilisant une méthode analogue à celle définie à la section 3.1. Puisque cette compilation est faite avant l'appel au méta-prédicat, le système est capable sans problème de connaître le contexte d'appel dans le quel doit être appelé la méta-donnée, pour obtenir un résultat analogue à celui qui est souhaité. Le méta-prédicat manipule alors une version compilée de la méta-données qu'il est possible d'exécuter à l'aide du prédicat `call/1`. Comme le système ne fournit aucun prédicat (hormis `call/1`) capable de manipuler ou créer une version compilée de méta-donnée, il préserve le masquage de l'implémentation.

Bien que cela ne soit pas clairement énoncé dans son manuel, pour contourner le problème de l'ordre supérieur, CIAO Prolog fait donc une distinction entre les termes et les objets d'ordre supérieur (les versions compilées de but). Cependant ces objets d'ordre supérieur ne sont pas explicitement des citoyens de première classe du langage. Nous préférerons une autre solution qui ne cache pas ces objets derrière une directive.

5.2.6 XSB

Le système XSB [23], qui est lui aussi considéré comme un système syntaxique, suit une approche assez différentes des systèmes précédents. En effet, il fait partie de la classes des systèmes fondés sur les atomes, au contraire des précédents qui sont eux fondés sur les prédicats. La principale différence est que XSB modularise aussi les symboles de fonction. Ainsi deux termes structurés construits dans deux modules différents ne pourront pas, par défaut, être unifiés. Il est toutefois possible d'importer, dans un module, des symboles publics d'un autre module, le système considérant alors que les modules partagent ces symboles. La sémantique du `call` est alors très simple, le méta-appel d'un terme correspond à l'appel au prédicat de même symbole et même arité, du module où le terme a été créé.

Cette solution ne fait cependant que déplacer le problème sur la construction dynamique de termes. En effet, dans XSB les termes construits à l'aide de `=../2`, `functor/3` et `read/1` sont tous supposés appartenir au module spécial `user`. Le système ne respecte alors pas l'indépendance de la stratégie de sélection (par exemple, dans un module distinct de `user`, `(functor(X,toto,1), X=toto(_))` échoue alors que `(X=toto(_), functor(X,toto,1))` réussi).

5.3 Notre Solution – Les Fermetures

Nous introduisons dans le langage des objets d'ordre supérieur, que nous appelons *fermetures*. Afin de créer et manipuler de telles données, nous avons ajouté deux nouveaux opérateurs `closure/3` et `apply/2`. L'expression `closure(X, G, C)` unifie `C` avec la fermeture construite à partir du but `G` dans lequel la variable `X` a été abstraite, tandis que `apply(C, T)` substitue dans la fermeture `C` la variable d'abstraction par le terme `T` et exécute le but obtenu.

Il faut noter qu'une fermeture ne peut être unifiée qu'avec des variables libres. De plus il est important de bien comprendre que `closure/3` est un nouvel opérateur et pas simplement un prédicat, notamment parce qu'il suppose d'une part que `G` soit un but (et pas une variable) à la compilation, et d'autre part que la variable d'abstraction soit libre dans le reste de la clause. Ce n'est pas le cas d'`apply/2` qui peut être vu comme un prédicat prédéfini.

Exemple 2. Pour illustrer l'utilisation des fermetures, réécrivons l'exemple précédent :

```
forall([], C).
forall([H| T], C):- apply(C, H),
                    forall(T, P).
```

Cette fois-ci, même si `toto/1` est un predicat privé d'un autre module, l'appel à `closure(X,toto(X),Z), forall([1,2,3],Z)` se comportera comme attendu.

Exemple 3. De plus les fermetures nous permettent de définir un nouvelle implémentation du `findall` garantissant la protection du code.

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

Par exemple les clauses retirées par l'appel `closure(X,retract(found(X)),Z), findall(C,L)` seront retirées proprement dans le module appellant, et non pas dans le module de définition du `findall`.

Cette solution présente plusieurs avantages :
- Dans une fermeture, il n'y a aucun test de permission dynamique a effectuer, car le prédicat est connu à la compilation
- les fermetures peuvent être compilées de façon efficace, car le système sait à la compilation que le deuxième argument de l'opérateur `closure` est un prédicat.

– La valeur du méta-paramètre est la même entre l'entrée et la sortie de l'appel (c.à.d. qu'il n'y a pas de conversion dynamique de données avant l'invocation d'un méta-prédicat).
– Comme le montre l'annexe A, `closure/3` et `apply/2` ont une implantation simple en programmation concurrente linéaire avec contraintes.

Parmi toutes les fermetures qui peuvent être créées à partir du langage, on peut en distinguer une particulièrement intéressante dans le contexte d'un système de modules fermé. Cette fermeture, que nous appelons *délégation*, est l'abstraction de `call/1`, c.à.d. la fermeture `Delegation` obtenue par application de l'opération `closure(X, call(X), Delegation)`. Grâce à cette construction un module peut rendre accessible, de façon explicite, son espace de travail à un module tiers. Par exemple en créant dans un module `m1` une délégation `Delegation` et en la passant à un module `m2` (par l'intermédiaire d'un prédicat public de `m2`), le module `m2` est capable d'exécuter n'importe quel but `G` dans l'espace de travail de `m1` à l'aide de l'appel `apply(Delegation, G)`.

Ainsi grâce à la combinaison de la méta-programmation et des fermetures, le langage obtenu est assez expressif pour permettre au programmeur de mettre en place différentes politiques de modules. Il peut choisir :

– Une politique ouverte, «à la SICStus», où l'espace de travail du module sera visible de l'extérieur : pour cela il suffit de déclarer le module public.
– Une politique fermée, où le module sera vu de l'extérieur comme une boîte noire : en n'utilisant aucune fermeture.
– Une politique intermédiaire, où certains modules auront accès à l'espace de travail et certains autres non : en passant, explicitement une délégation aux modules que qui pourront accéder à l'espace de travail.

6 Conclusion

Dans son article [27], Warren critiquait les extensions de Prolog par des mécanismes d'ordre supérieur (et 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 avons montré que ce n'était plus le cas dans le cadre d'un système modulaire fermé.

Le système de modules que nous avons proposé est proche de celui de CIAO Prolog, dans son implantation, mais présente l'avantage de dégager le concept de fermeture et de le positionner par rapport aux prédicats de méta-programmation dans le cadre d'un sémantique formelle.

Comme preuve du concept, le système de modules a été implanté en GNU-Prolog et plusieurs développements de bibliothèques ont été réalisés dans l'équipe à partir de la version modulaire de GNU Prolog RH, pour le développement de SiLCC. On peut citer, entre autres, un lexeur-analyseur syntaxique conçu par De Rauglaudre permettant des extensions de syntaxe plus générales que la traditionnelle directive `op/3`, et relatives aux modules. Cette bibliothèque fait partie du bootstrap du langage SiLCC. Par ailleurs, le portage par Bouissou de la librairie CHR de Schrijvers fournit un exemple d'utilisation intensive du système de modules.

Remerciements

Nous tenons à remercier Sumit Kumar pour le travail préliminaire qu'il a réalisé sur le sujet pendant son stage à l'INRIA durant l'été 2003. Nous remercions également Emmanuel Coquery et Sylvain Soliman pour les discussions fructueuses que nous avons eues ensemble sur ce sujet. Nous sommes aussi reconnaissants envers Daniel de Rauglaudre et Olivier Bouissou pour les commentaires qu'ils nous ont apportés sur l'implantation réalisée.

Références

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

[2] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Logic Programming. MIT Press, 1991.

[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. 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.

[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 and Philipe Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 6, October 2001.

[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. *Programmation Logique par Contraintes*. Collection Cours de l'Ecole Polytechnique. Ed. Ellipses, Paris (192p), 1996.

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

[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 13211-2 :1995.

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

[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] 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.

[26] 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.

[27] 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.

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

A Fermeture en LCC

Nous proposons ici une définition de *closure*/3 et *apply*/3 dans la programmation concurrente linéaire avec contraintes (LCC). En fait ces deux construction peuvent être vu simplement comme du sucre syntaxique :

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

Dans la LCC, par définition le coté doit de l'opérateur «ask» (c.à.d. $\rightarrow$) doit être un agent (la notion d'agent est équivalent à celle de but dans la LCC), ainsi A doit toujours être instancié à la compilation.