

Constraint Logic Programming

Sylvain Soliman and François Fages
{Sylvain.Soliman,Francois.Fages}@inria.fr

INRIA – Project-Team CONTRAINTES

MPRI C-2-4-1 Course – September 2009 - February 2010

Part III: CLP - Operational and Fixpoint Semantics

- 1 Operational Semantics
 - CSLD resolution
 - Observables
- 2 Fixpoint Semantics
 - Fixpoint Preliminaries
 - Fixpoint Semantics of Successes
 - Fixpoint Semantics of Computed Answers
- 3 Program Analysis
 - Abstract Interpretation
 - Constraint-based Model Checking

Part IV: Logical Semantics

- 4 Logical Semantics of CLP(\mathcal{X})
 - Soundness
 - Completeness
- 5 Automated Deduction
 - Proofs in Group Theory
- 6 CLP(λ)
 - λ -calculus
 - Proofs in λ -calculus
- 7 Negation as Failure
 - Finite Failure
 - Clark's Completion
 - Soundness w.r.t. Clark's Completion
 - Completeness w.r.t. Clark's Completion

Part V: Practical CLP Programming

- 8 CLP implementation, the WAM
- 9 Optimizing CLP
- 10 Summing up

Part VI: Concurrent Constraint Programming

11 Introduction

- Syntax
- CC vs. CLP

12 Operational Semantics

- Transitions
- Properties
- Observables

13 Examples

- append
- merge
- $CC(\mathcal{FD})$

Part VII: CC - Denotational Semantics

- 14 Deterministic Case
 - Syntax
 - I/O Function
 - Terminal Stores
- 15 Constraint Propagation
 - Closure Operators
 - Chaotic Iteration
- 16 Non-deterministic Case
 - Problems
 - Blind Choice
 - Example: `merge`
- 17 Sequentiality

Part VIII: CC and Linear Logic

18 CC - Logical Semantics

- Intuitionistic
- Linear
- Soundness
- Completeness

19 Must Properties

- Definition
- Soundness
- Completeness

20 Program Analysis

- Equivalence
- Phase Semantics and Theorem Proving

Part IX

LCC

Part IX: LCC

- 21 LCC
 - Syntax
 - Operational Semantics
- 22 Examples
 - Dining Philosophers
 - Other examples
 - Indexicals
- 23 Logical Semantics
 - Intuitionistic Linear Logic
 - Phase Semantics
 - Example
- 24 Modules
 - First class closures
 - Encoding modules
 - Code protection

Linear Constraint Systems $(\mathcal{C}, \vdash_{\mathcal{C}})$

\mathcal{C} is a set of formulas built from V, Σ with logical operators: $1, \otimes, \exists$ and $!$;

$\vdash_{\mathcal{C}} \subseteq \mathcal{C} \times \mathcal{C}$ defines the non-logical axioms of the constraint system.

$\vdash_{\mathcal{C}}$ is the least subset of $\mathcal{C}^* \times \mathcal{C}$ containing $\vdash_{\mathcal{C}}$ and closed by:

$$\begin{array}{c}
 c \vdash c \quad \frac{\Gamma, c \vdash d \quad \Delta \vdash c}{\Gamma, \Delta \vdash d} \vdash 1 \quad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c} \\
 \\
 \frac{\Gamma \vdash c_1 \quad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \quad \frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \quad \frac{\Gamma \vdash c[t/x]}{\Gamma \vdash \exists x c} \quad \frac{\Gamma, c \vdash d}{\Gamma, \exists x c \vdash d} \quad x \notin \text{fv}(\Gamma, d) \\
 \\
 \frac{\Gamma, c \vdash d}{\Gamma, !c \vdash d} \quad \frac{! \Gamma \vdash d}{! \Gamma \vdash !d} \quad \frac{\Gamma \vdash d}{\Gamma, !c \vdash d} \quad \frac{\Gamma, !c, !c \vdash d}{\Gamma, !c \vdash d}
 \end{array}$$

A **synchronization constraint** is a constraint not appearing in $\vdash_{\mathcal{C}}$

Same agents and observables as CC

Processes $P ::= \mathcal{D}.A$

Declarations $\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$

Agents $A ::= \text{tell}(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid A + A \mid \exists xA \mid p(\vec{x})$

- observing the set of **success stores**,

$$\mathcal{O}_{ss}(\mathcal{D}.A; c) = \{\exists \vec{x}d \in \mathcal{C} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \epsilon)\}$$

- observing the set of **terminal stores** (successes and suspensions),

$$\mathcal{O}_{ts}(\mathcal{D}.A; c) = \{\exists \vec{x}d \in \mathcal{C} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \Gamma) \dashv\rightarrow\}$$

- observing the set of **accessible stores**,

$$\mathcal{O}_{as}(\mathcal{D}.A; c) = \{\exists \vec{x}d \in \mathcal{C} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \Gamma)\}$$

Linear-CC(\mathcal{C}) Transitions

Tell $(X; c; \text{tell}(d), \Gamma) \longrightarrow (X; c \otimes d; \Gamma)$

Ask
$$\frac{c \vdash_c d \otimes e[\vec{t}/\vec{y}]}{(X; c; \forall \vec{y}(e \rightarrow A), \Gamma) \longrightarrow (X; d; A[\vec{t}/\vec{y}], \Gamma)}$$

Hiding
$$\frac{y \notin X \cup \text{fv}(c, \Gamma)}{(X; c; \exists y A, \Gamma) \longrightarrow (X \cup \{y\}; c; A, \Gamma)}$$

Call
$$\frac{(p(\vec{y}) = A) \in \mathcal{D}}{(X; c; p(\vec{y}), \Gamma) \longrightarrow (X; c; A, \Gamma)}$$

Choice $(X; c; A + B, \Gamma) \longrightarrow (X; c; A, \Gamma)$
 $(X; c; A + B, \Gamma) \longrightarrow (X; c; B, \Gamma)$

Congr.
$$\frac{z \notin \text{fv}(A)}{\exists y A \equiv \exists z A[z/y]} \quad A \parallel B \equiv B \parallel A \quad A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$$

An LCC(\mathcal{FD}) program for the dining philosophers

Goal(N) = RecPhil(1,N).

RecPhil(M,P) =

$M \neq P \rightarrow (\text{Philo}(M,P) \parallel \text{fork}(M) \parallel \text{RecPhil}(M+1,P))$
 \parallel
 $M = P \rightarrow (\text{Philo}(M,P) \parallel \text{fork}(M)).$

Philo(I,N) =

$(\text{fork}(I) \otimes \text{fork}(I+1 \bmod N)) \rightarrow$
 $(\text{eat}(I) \parallel$
 $\text{eat}(I) \rightarrow (\text{fork}(I) \parallel \text{fork}(I+1 \bmod N) \parallel$
 $\text{Philo}(I,N)).$

Safety properties: deadlock freeness, two neighbors don't eat at the same time, etc.

Producer Consumer Protocol in LCC

$$P = \text{dem} \rightarrow (\text{pro} \parallel P)$$
$$C = \text{pro} \rightarrow (\text{dem} \parallel C)$$
$$\text{init} = \text{dem}^n \parallel P^m \parallel C^k$$

Deadlock-freeness: $\text{init} \not\rightarrow_{LCC} \text{dem}^{n'} \parallel P^{m'} \parallel C^{k'} \parallel \text{pro}^{l'}$, with either $n' = l' = 0$ or $m' = 0$ or $k' = 0$

Number of units consumed always $<$ number of units produced:

$$P = \text{dem} \rightarrow (\text{pro} \parallel P \parallel$$
$$\quad \forall X (\text{count}(\text{np}, X) \rightarrow \text{count}(\text{np}, X+1)))$$
$$\text{init} = \text{dem}^n \parallel P^m \parallel C^k \parallel \text{np}=0 \parallel \text{nc}=0$$
$$\text{init} \not\rightarrow_{LCC} \text{dem}^{n'} \parallel \text{pro}^{l'} \parallel P^m \parallel C^k \parallel \text{np}=\text{np}_0 \parallel \text{nc}=\text{nc}_0$$

with $\text{nc}_0 > \text{np}_0$

$CC(\mathcal{FD})$ in $LCC(\mathcal{H})$

$CC(\mathcal{FD})$ propagators, including **indexicals**, are now easily embedded in LCC.

$fd(X) =$

CC(\mathcal{FD}) in LCC(\mathcal{H})

CC(\mathcal{FD}) propagators, including **indexicals**, are now easily embedded in LCC.

$\text{fd}(X) = \text{tell}(\text{min}(X, \text{min_integer}) \otimes \text{max}(X, \text{max_integer}))$

' $x \geq_1 y + c$ '(X, Y, C) =

CC(\mathcal{FD}) in LCC(\mathcal{H})

CC(\mathcal{FD}) propagators, including **indexicals**, are now easily embedded in LCC.

$\text{fd}(X) = \text{tell}(\text{min}(X, \text{min_integer}) \otimes \text{max}(X, \text{max_integer}))$

$'x \geq_1 y + c'(X, Y, C) =$
 $\text{min}(X, \text{MinX}) \otimes \text{min}(Y, \text{MinY}) \otimes (\text{MinX} < \text{MinY} + C)$
 $\rightarrow (\text{tell}(\text{min}(X, \text{MinY} + C) \otimes \text{min}(Y, \text{MinY}))$
 $\parallel 'x \geq_1 y + c'(X, Y, C))$

$'x \geq y + c'(X, Y, C) =$

CC(\mathcal{FD}) in LCC(\mathcal{H})

CC(\mathcal{FD}) propagators, including **indexicals**, are now easily embedded in LCC.

$\text{fd}(X) = \text{tell}(\text{min}(X, \text{min_integer}) \otimes \text{max}(X, \text{max_integer}))$

$'x \geq_1 y + c'(X, Y, C) =$
 $\text{min}(X, \text{MinX}) \otimes \text{min}(Y, \text{MinY}) \otimes (\text{MinX} < \text{MinY} + C)$
 $\rightarrow (\text{tell}(\text{min}(X, \text{MinY} + C) \otimes \text{min}(Y, \text{MinY}))$
 $\parallel 'x \geq_1 y + c'(X, Y, C))$

$'x \geq y + c'(X, Y, C) = 'x \geq_1 y + c'(X, Y, C) \parallel 'x \geq_2 y + c'(X, Y, C)$

$'\text{ask}(x \geq y) \rightarrow a'(X, Y, A) =$

CC(\mathcal{FD}) in LCC(\mathcal{H})

CC(\mathcal{FD}) propagators, including **indexicals**, are now easily embedded in LCC.

$$\text{fd}(X) = \text{tell}(\text{min}(X, \text{min_integer}) \otimes \text{max}(X, \text{max_integer}))$$

$$\begin{aligned} 'x \geq_1 y + c'(X, Y, C) &= \\ &\text{min}(X, \text{MinX}) \otimes \text{min}(Y, \text{MinY}) \otimes (\text{MinX} < \text{MinY} + C) \\ &\rightarrow (\text{tell}(\text{min}(X, \text{MinY} + C) \otimes \text{min}(Y, \text{MinY})) \\ &\quad \parallel 'x \geq_1 y + c'(X, Y, C)) \end{aligned}$$

$$'x \geq y + c'(X, Y, C) = 'x \geq_1 y + c'(X, Y, C) \parallel 'x \geq_2 y + c'(X, Y, C)$$

$$\begin{aligned} 'ask(x \geq y) \rightarrow a'(X, Y, A) &= \\ &\text{min}(X, \text{MinX}) \otimes \text{max}(Y, \text{MaxY}) \otimes (\text{MinX} > \text{MaxY}) \\ &\rightarrow A \parallel \text{tell}(\text{min}(X, \text{MinX}) \otimes \text{max}(Y, \text{MaxY})) \end{aligned}$$

Imperative variables allow a finer control, which is necessary for certain constraint solvers, see for instance the implementation of a Simplex solver in LCC [Sch99].

Logical Semantics

Simple translation of LCC into ILL:

$$\begin{array}{ll} \text{tell}(c)^\dagger = c & p(\vec{x})^\dagger = p(\vec{x}) \\ \forall \vec{y}(c \rightarrow A)^\dagger = \forall \vec{y}(c \multimap A^\dagger) & (A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \\ (A + B)^\dagger = A^\dagger \& B^\dagger & (\exists xA)^\dagger = \exists xA^\dagger \end{array}$$

ILL(\mathcal{C}, \mathcal{D}) denotes the deduction system obtained by adding to intuitionistic linear logic the axioms:

- $c \vdash d$ for every $c \Vdash_{\mathcal{C}} d$ in $\Vdash_{\mathcal{C}}$,
- $p(\vec{x}) \vdash A^\dagger$ for every declaration $p(\vec{x}) = A$ in \mathcal{D} .

Same soundness/completeness results as for CC.

Phase Semantics

A **phase space** $\mathbf{P} = \langle P, \times, 1, \mathcal{F} \rangle$ is a structure such that:

- 1 $\langle P, \times, 1 \rangle$ is a commutative monoid.
- 2 the set of facts \mathcal{F} is a subset of $\mathcal{P}(P)$ such that: \mathcal{F} is closed by arbitrary intersection, and for all $A \subset P$, for all $F \in \mathcal{F}$,
 $A \multimap F \triangleq \{x \in P : \forall a \in A, a \times x \in F\}$ is a fact.

We define the following operations:

$$A \& B \triangleq A \cap B$$

$$A \otimes B \triangleq \bigcap \{F \in \mathcal{F} : A \times B \subset F\} \quad A \oplus B \triangleq \bigcap \{F \in \mathcal{F} : A \cup B \subset F\}$$

$$\exists x A \triangleq \bigcap \{F \in \mathcal{F} : (\bigcup_x A) \subset F\} \quad \forall x A \triangleq \bigcap \{F \in \mathcal{F} : (\bigcap_x A) \subset F\}$$

We'll note $\top \triangleq P$, $\mathbf{0} \triangleq \bigcap \{F \in \mathcal{F}\}$ and $\mathbf{1} \triangleq \bigcap \{F \in \mathcal{F} \mid 1 \in F\}$.

Interpretation

Let η be a valuation assigning a fact to each atomic formula such that: $\eta(\top) = \top$, $\eta(\mathbf{1}) = \mathbf{1}$ and $\eta(\mathbf{0}) = \mathbf{0}$.

We can now define inductively the interpretation of a sequent:

$$\eta(\Gamma \vdash A) = \eta(\Gamma) \multimap \eta(A) \quad \eta(\Gamma) = \mathbf{1} \text{ if } \Gamma \text{ is empty}$$

$$\eta(\Gamma, \Delta) = \eta(\Gamma) \otimes \eta(\Delta) \quad \eta(A \otimes B) = \eta(A) \otimes \eta(B)$$

$$\eta(A \& B) = \eta(A) \& \eta(B) \quad \eta(A \multimap B) = \eta(A) \multimap \eta(B)$$

We then define the notion of validity as follows:

$\mathbf{P}, \eta \models (\Gamma \vdash A)$ iff $\mathbf{1} \in \eta(\Gamma \vdash A)$, thus $\eta(\Gamma) \subset \eta(A)$.

Soundness:

$$\Gamma \vdash_{ILL} A \text{ implies } \forall \mathbf{P}, \forall \eta, \mathbf{P}, \eta \models (\Gamma \vdash A).$$

(syntactic proof for completeness)

Phase Counter-Models

We impose to every valuation η to satisfy the non-logical axioms of $ILL_{\mathcal{C}, \mathcal{D}}$:

- $\eta(c) \subset \eta(d)$ for every $c \Vdash_{\mathcal{C}} d$ in $\Vdash_{\mathcal{C}}$,
- $\eta(p) \subset \eta(A^\dagger)$ for every declaration $p = A$ in \mathcal{D} .

The contrapositive of the two soundness theorems becomes:

Theorem 1

to prove a safety property of the form

$$(X; c; A) \not\vdash (Y; d; B)$$

It is enough to show

$$\exists \mathbf{P}, \exists \eta, \exists a \in \eta((X; c; A)^\dagger) \text{ such that } a \notin \eta((Y; d; B)^\dagger).$$

Producer Consumer Protocol in LCC

$$P = \text{dem} \rightarrow (\text{pro} \parallel P)$$

$$C = \text{pro} \rightarrow (\text{dem} \parallel C)$$

$$\text{init} = \text{dem}^n \parallel P^m \parallel C^k$$

Deadlock-freeness: $\text{init} \not\rightarrow \text{dem}^{n'} \parallel P^{m'} \parallel C^{k'} \parallel \text{pro}^{l'}$, with either $n' = l' = 0$ or $m' = 0$ or $k' = 0$

Producer Consumer Protocol in LCC

$$P = \text{dem} \rightarrow (\text{pro} \parallel P)$$

$$C = \text{pro} \rightarrow (\text{dem} \parallel C)$$

$$\text{init} = \text{dem}^n \parallel P^m \parallel C^k$$

Deadlock-freeness: $\text{init} \not\rightarrow \text{dem}^{n'} \parallel P^{m'} \parallel C^{k'} \parallel \text{pro}^{l'}$, with either $n' = l' = 0$ or $m' = 0$ or $k' = 0$

Let us consider the structure $(\mathbb{N}, \times, 1, \mathcal{P}(\mathbb{N}))$, it is obviously a phase space.

Producer Consumer Protocol in LCC

$$P = \text{dem} \rightarrow (\text{pro} \parallel P)$$

$$C = \text{pro} \rightarrow (\text{dem} \parallel C)$$

$$\text{init} = \text{dem}^n \parallel P^m \parallel C^k$$

Deadlock-freeness: $\text{init} \not\rightarrow \text{dem}^{n'} \parallel P^{m'} \parallel C^{k'} \parallel \text{pro}^{l'}$, with either $n' = l' = 0$ or $m' = 0$ or $k' = 0$

Let us consider the structure $(\mathbb{N}, \times, 1, \mathcal{P}(\mathbb{N}))$, it is obviously a phase space.

Let us define the following valuation:

$$\eta(P) = \{2\} \quad \eta(C) = \{3\} \quad \eta(\text{dem}) = \{5\} \quad \eta(\text{pro}) = \{5\}$$

$$\eta(\text{init}) = \{2^m \cdot 3^k \cdot 5^n\}$$

Proof

- We have to check the correctness of η :
 $\forall p_1 \in \eta(P), \exists p_2 \in \eta(P), dem \cdot p_1 = pro \cdot p_2$, hence
 $\eta(P) \subset \eta(\text{body of } P)$.
The same for C , and $\eta(\text{init}) = \eta(\text{body of init})$.
- Instead of exhibiting a counter-example, we will prove *Ab absurdum* that the inclusion
 $\eta(\text{init}) \subset \eta(\text{dem}^{n'} \parallel P^{m'} \parallel C^{k'} \parallel \text{pro}^{l'})$ is impossible.
Suppose $\eta(\text{init}) \subset \{5^{n'} \cdot 2^{m'} \cdot 3^{k'} \cdot 5^{l'}\}$ Comparing the power
of 5, 3 and 2, anything else than: $n' + l' = n$ and $m' = m$ and
 $k' = k$ is impossible, and therefore if there is a deadlock
($n' + l' = 0 \neq n$, or $m' = 0 \neq m$, or $k' = 0 \neq k$) $\eta(\text{init})$ is
not a subset of its interpretation and thus init does not
reduce into it, qed.

Automatization

The search for a phase space can be automatized, if one accepts some restrictions:

- always use the structure $(\mathbb{N}, \times, 1, \mathcal{P}(\mathbb{N}))$;

Automatization

The search for a phase space can be automatized, if one accepts some restrictions:

- always use the structure $(\mathbb{N}, \times, 1, \mathcal{P}(\mathbb{N}))$;
[*be careful that integers are invertible*]

Automatization

The search for a phase space can be automatized, if one accepts some restrictions:

- always use the structure $(\mathbb{N}, \times, 1, \mathcal{P}(\mathbb{N}))$;
[*be careful that integers are invertible*]
- always look for simple (singleton/doubleton/finite) interpretations.

Automatization

The search for a phase space can be automatized, if one accepts some restrictions:

- always use the structure $(\mathbb{N}, \times, 1, \mathcal{P}(\mathbb{N}))$;
[*be careful that integers are invertible*]
- always look for simple (singleton/doubleton/finite) interpretations.
[*might lead to confusions*]

Declarations as agents

Processes $P ::= D.A$

Declarations $\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$

Agents $A ::= \text{tell}(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists xA \mid A + A \mid p(\vec{x})$

becomes

Processes $A ::= \text{tell}(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists xA \mid \forall \vec{x}(c \Rightarrow A)$

Operational semantics of **persistent asks** is the same as that of asks except that the agent is not consumed.

Local choice can be encoded through asks:

$A + B =$

Declarations as agents

Processes $P ::= D.A$

Declarations $\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$

Agents $A ::= \text{tell}(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists xA \mid A + A \mid p(\vec{x})$

becomes

Processes $A ::= \text{tell}(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists xA \mid \forall \vec{x}(c \Rightarrow A)$

Operational semantics of **persistent asks** is the same as that of asks except that the agent is not consumed.

Local choice can be encoded through asks:

$$A + B = \exists x(\text{tell}(\text{choice}(x)) \parallel \text{choice}(x) \rightarrow A \parallel \text{choice}(x) \rightarrow B)$$

Closures as persistent asks

A closure is simply some code with an environment. The persistent ask and the hiding mechanism provide just that.

forall iterator

$$\begin{aligned} \text{forall}([\] &\Rightarrow \text{tell}(\text{true}) \parallel \\ \forall H, T \text{ forall}([H|T] &\Rightarrow \text{tell}(\text{apply}(H)) \parallel \text{tell}(\text{forall}(T)) \parallel \\ \forall x(\text{apply}(x) &\Rightarrow \text{Body}(x)) \end{aligned}$$

This idea provides a simple encoding of declarations, but also of multi-headed (CHR) rules as agents.

Observables definition leads to **separating** the constraints in order to project “process calls” and distinguish declarations from usual suspensions.

Modules as closures

The closure mechanism provides a natural encoding of modules as **first class** citizens of LCC by simply considering the first argument of predicates as “module name”.

Can be used for CLP too (see [HF06]) with better properties w.r.t. meta-predicates than usual module systems (e.g. SICStus)

The scope of module declarations is given by the scope of the corresponding variable.

There are two problems however with this module system :

- unification

Modules as closures

The closure mechanism provides a natural encoding of modules as **first class** citizens of LCC by simply considering the first argument of predicates as “module name”.

Can be used for CLP too (see [HF06]) with better properties w.r.t. meta-predicates than usual module systems (e.g. SICStus)

The scope of module declarations is given by the scope of the corresponding variable.

There are two problems however with this module system :

- unification \Rightarrow union of clauses;

Modules as closures

The closure mechanism provides a natural encoding of modules as **first class** citizens of LCC by simply considering the first argument of predicates as “module name”.

Can be used for CLP too (see [HF06]) with better properties w.r.t. meta-predicates than usual module systems (e.g. SICStus)

The scope of module declarations is given by the scope of the corresponding variable.

There are two problems however with this module system :

- unification \Rightarrow union of clauses;
- module name capture with \forall

Code protection

To enforce code protection a simple technique is to restrict the syntax and the constraint system:

- No universal quantification on module variables (MLCC)
- No constraints making “all variables equal”

If we enforce the second one by imposing that $\{x, y\} \subset fv(c)$ whenever $c \vdash_c x = y \otimes \top$, we get :

Theorem 2 (Code protection [HFS07])

Let A and B be two MLCC agents. If A has no inner module and y is used in A and B only in modular tells of the form $y : l$ with $y \notin fv(l)$, then A is protected in $\exists y(y\{A\} \parallel B)$.

SICStus modules do not offer code protection

```
:- module(library, [mycall/1]).  
  
p :- write('library:p/0  ').  
  
:- meta_predicate(mycall(:)).  
mycall(M:G) :- M:p, call(M:G).
```

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

Unlimited qualification.

The meta-predicate declaration even allows for dynamic qualification.

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

ECLiPSe modules do not either

```
:- module(library, [mycall/1]).  
  
p :- write('library:p/0 ').  
  
:- tool(mycall/1, mycall/2).  
mycall(G, M) :- call(p)@M, call(G)@M.
```

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

Only exported predicates accessible through qualification, but unlimited call@ construct.

The tool declaration allows for dynamic qualification.

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

Bibliography I



Rémy Haemmerlé and François Fages.

Modules for Prolog revisited.

In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.



Rémy Haemmerlé, François Fages, and Sylvain Soliman.

Closures and modules within linear logic concurrent constraint programming.

In V. Arvind and Sanjiva Prasad, editors, *Proceedings of FSTTCS 2007, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 544–556. Springer-Verlag, 2007.



Vincent Schächter.

Programmation concurrente avec contraintes fondée sur la logique linéaire.

PhD thesis, Université d'Orsay, Paris 11, 1999.