Constraint Logic Programming

Sylvain Soliman
Sylvain.Soliman@inria.fr

Project-Team LIFEWARE

MPRI 2.35.1 Course – September–November 2017

Part I: CLP - Introduction and Logical Background

Part II: Constraint Logic Programs

6. Constraint Languages
7. CLP($\mathcal{X}$)
8. CLP($\mathcal{H}$)
9. CLP($\mathcal{R}, \mathcal{FD}, \mathcal{B}$)

Part III: CLP - Operational and Fixpoint Semantics

Part IV: Logical Semantics

Part V: Constraint Solving

Part VI: Practical CLP Programming

Part VII: More Constraint Programming

Part VIII: Programming Project

Part IX: Concurrent Constraint Programming

Part X: CC - Denotational Semantics

Part XI: CC and Linear Logic

Intuitionistic Linear Logic

Multiplicatives

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Delta, \Gamma, A \multimap B \vdash C} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$$

Additives

$$\frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}$$

Constants

$$\frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A} \qquad \vdash \mathbf{1} \qquad \bot \vdash \qquad \frac{\Gamma \vdash}{\Gamma \vdash \bot} \qquad \Gamma \vdash \top \qquad \Gamma, \mathbf{0} \vdash A$$

ILL = the Logic of CC agents

Translation:

$(A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \qquad (c \to A)^\dagger =$

ILL = the Logic of CC agents

Translation:

$(A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \qquad (c \to A)^\dagger = c \multimap A^\dagger \qquad \mathit{tell}(c)^\dagger =$

ILL = the Logic of CC agents

Translation:

$(A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \qquad (c \to A)^\dagger = c \multimap A^\dagger \qquad \mathit{tell}(c)^\dagger = !c$

$(A + B)^\dagger =$

ILL = the Logic of CC agents

Translation:

$$(A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \qquad (c \to A)^\dagger = c \multimap A^\dagger \qquad \mathit{tell}(c)^\dagger = !c$$
$$(A + B)^\dagger = A^\dagger \,\&\, B^\dagger \qquad (\exists x A)^\dagger = \exists x A^\dagger \qquad p(\vec{x})^\dagger = p(\vec{x})$$
$$(X; c; \Gamma)^\dagger = \exists X(!c \otimes \Gamma^\dagger)$$

Axioms: $!c \vdash !d$ for all $c \vdash_{\mathcal{C}} d$ $\qquad$ $p(\vec{x}) \vdash A^\dagger$ for all $p(\vec{x}) = A \in \mathcal{D}$

ILL = the Logic of CC agents

Translation:

$$(A \parallel B)^\dagger = A^\dagger \otimes B^\dagger \qquad (c \to A)^\dagger = c \multimap A^\dagger \qquad tell(c)^\dagger = !c$$
$$(A + B)^\dagger = A^\dagger \,\&\, B^\dagger \qquad (\exists x A)^\dagger = \exists x A^\dagger \qquad p(\vec{x})^\dagger = p(\vec{x})$$
$$(X; c; \Gamma)^\dagger = \exists X(!c \otimes \Gamma^\dagger)$$

Axioms: $!c \vdash !d$ for all $c \vdash_{\mathcal{C}} d$ $\qquad$ $p(\vec{x}) \vdash A^\dagger$ for all $p(\vec{x}) = A \in \mathcal{D}$

Soundness and Completeness

If $(c; \Gamma) \longrightarrow_{CC} (d; \Delta)$ then $c^\dagger \otimes \Gamma^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} d^\dagger \otimes \Delta^\dagger$

If $A^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c$ then *there exists a success store* d such that $(true; A) \longrightarrow_{CC} (d; \emptyset)$ and $d \vdash_{\mathcal{C}} c$

If $A^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top$ then *there exists an accessible store* d such that $(true; A) \longrightarrow_{CC} (d; \Gamma)$ and $d \vdash_{\mathcal{C}} c$

Part XII: LCC

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

One can now easily embed in LCC our CC($\mathcal{FD}$) propagators, including

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

One can now easily embed in LCC our CC($\mathcal{FD}$) propagators, including indexicals.

```
fd(X) =
```

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

One can now easily embed in LCC our CC($\mathcal{FD}$) propagators, including indexicals.

```
fd(X) = tell(min(X,min_integer) ⊗ max(X,max_integer))
```

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

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

One can now easily embed in LCC our CC($\mathcal{FD}$) propagators, including indexicals.

```
fd(X) = tell(min(X,min_integer) ⊗ max(X,max_integer))

'x≥₁y+c'(X,Y,C) =
    min(X,MinX) ⊗ min(Y,MinY) ⊗ (MinX<MinY+C)
    → (tell(min(X,MinY+C) ⊗ min(Y,MinY))
       ∥ 'x≥₁y+c'(X,Y,C))

'x≥y+c'(X,Y,C) =
```

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

One can now easily embed in LCC our CC($\mathcal{FD}$) propagators, including indexicals.

```
fd(X) = tell(min(X,min_integer) ⊗ max(X,max_integer))

'x≥₁y+c'(X,Y,C) =
    min(X,MinX) ⊗ min(Y,MinY) ⊗ (MinX<MinY+C)
    → (tell(min(X,MinY+C) ⊗ min(Y,MinY))
       ‖ 'x≥₁y+c'(X,Y,C))

'x≥y+c'(X,Y,C) = 'x≥₁y+c'(X,Y,C) ‖ 'x≥₂y+c'(X,Y,C)

'ask(x≥y)→a'(X,Y,A) =
```

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

One can now easily embed in LCC our CC($\mathcal{FD}$) propagators, including indexicals.

```
fd(X) = tell(min(X,min_integer) ⊗ max(X,max_integer))

'x≥1y+c'(X,Y,C) =
    min(X,MinX) ⊗ min(Y,MinY) ⊗ (MinX<MinY+C)
    → (tell(min(X,MinY+C) ⊗ min(Y,MinY))
       ∥ 'x≥1y+c'(X,Y,C))

'x≥y+c'(X,Y,C) = 'x≥1y+c'(X,Y,C) ∥ 'x≥2y+c'(X,Y,C)

'ask(x≥y)→a'(X,Y,A) =
    min(X,MinX) ⊗ max(Y,MaxY) ⊗ (MinX≥MaxY)
    → A ∥ tell(min(X,MinX) ⊗ max(Y,MaxY))
```

Imperative variables allow a finer control, which is necessary for certain constraint solvers, e.g. the implementation of a Simplex solver in LCC [Schachter99these]

Part XIII

LCC Logical Semantics and more

Part XIII: LCC Logical Semantics and more

Logical Semantics

Simple translation of LCC into ILL:

$$tell(c)^{\dagger} =$$

Logical Semantics

Simple translation of LCC into ILL:

$$tell(c)^\dagger = c \qquad\qquad (A \parallel B)^\dagger = A^\dagger \otimes B^\dagger$$
$$\forall \vec{y}(c \rightarrow A)^\dagger =$$

Logical Semantics

Simple translation of LCC into ILL:

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

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:

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

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

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

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:

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

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

$$\eta(A \,\&\, B) = \eta(A) \,\&\, \eta(B) \qquad \eta(A \multimap B) = \eta(A) \multimap \eta(B)$$

We then define the notion of validity as follows:

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

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:

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

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

$$\eta(A \,\&\, B) = \eta(A) \,\&\, \eta(B) \qquad \eta(A \multimap B) = \eta(A) \multimap \eta(B)$$

We then define the notion of validity as follows:
$\mathbf{P}, \eta \models (\Gamma \vdash A)$ iff $1 \in \eta(\Gamma \vdash A)$, 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 $\mathrm{ILL}_{\mathcal{C},\mathcal{D}}$:

- $\eta(c) \subset \eta(d)$ 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

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

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

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

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

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

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

Producer Consumer Protocol in LCC

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

Deadlock-freeness: $\texttt{init} \not\longrightarrow \texttt{dem}^{n'} \parallel \texttt{P}^{m'} \parallel \texttt{C}^{k'} \parallel \texttt{pro}^{l'}$, 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

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

Deadlock-freeness: $\mathtt{init} \not\longrightarrow \mathtt{dem}^{n'} \parallel \mathtt{P}^{m'} \parallel \mathtt{C}^{k'} \parallel \mathtt{pro}^{l'}$, 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:

$$\eta(\mathtt{P}) = \{2\} \;\; \eta(\mathtt{C}) = \{3\} \;\; \eta(\mathtt{dem}) = \{5\} \;\; \eta(\mathtt{pro}) = \{5\}$$

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

Proof

We have to check the correctness of η:
$\forall p_1 \in \eta(\mathtt{P}), \exists p_2 \in \eta(\mathtt{P}), \mathit{dem} \cdot p_1 = \mathit{pro} \cdot p_2$,
hence $\eta(\mathtt{P}) \subset \eta($body of $\mathtt{P})$.
The same for $\mathtt{C}$, and $\eta(\mathtt{init}) = \eta($body of $\mathtt{init})$.

Instead of exhibiting a counter-example, we prove *Ab absurdum* the impossibility of the inclusion

$$\eta(\mathtt{init}) \subset \eta(\mathtt{dem}^{n'} \parallel \mathtt{P}^{m'} \parallel \mathtt{C}^{k'} \parallel \mathtt{pro}^{l'})$$

Proof (cont.)

Suppose $\eta(\texttt{init}) \subset \{5^{n'} \cdot 2^{m'} \cdot 3^{k'} \cdot 5^{l'}\}$

Since $\eta(\texttt{init}) = \{2^m \cdot 3^k \cdot 5^n\}$
anything else than: $n' + l' = n$ and $m' = m$ and $k' = k$ is impossible

now note that if there is a deadlock we have:
$n' + l' = 0 \neq n$, or $m' = 0 \neq m$, or $k' = 0 \neq k$

$\eta(\texttt{init})$ is thus not a subset of the interpretation of any deadlock and thus `init` does not reduce into it, $\square$

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	$P ::= \mathcal{D}.A$
Declarations	$\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$
Agents	$A ::= tell(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists x A \mid A + A \mid p(\vec{x})$

becomes

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

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

Local choice

Declarations as agents

Processes	$P ::= \mathcal{D}.A$
Declarations	$\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$
Agents	$A ::= tell(c) \mid \forall\vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists x A \mid A + A \mid p(\vec{x})$

becomes

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

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	$P ::= \mathcal{D}.A$
Declarations	$\mathcal{D} ::= p(\vec{x}) = A, \mathcal{D} \mid \epsilon$
Agents	$A ::= tell(c) \mid \forall \vec{x}(c \rightarrow A) \mid A \parallel A \mid \exists x A \mid A + A \mid p(\vec{x})$

becomes

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

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:

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

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

$$forall([]) \Rightarrow tell(true) \parallel$$
$$\forall H, T\ forall([H|T]) \Rightarrow tell(apply(H)) \parallel tell(forall(T)) \parallel$$
$$\forall x(apply(x) \Rightarrow Body(x))$$

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

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

$$forall([]) \Rightarrow tell(true) \parallel$$
$$\forall H, T\ forall([H|T]) \Rightarrow tell(apply(H)) \parallel tell(forall(T)) \parallel$$
$$\forall x(apply(x) \Rightarrow Body(x))$$

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

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 [HF06iclp]) 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 [HF06iclp]) 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 [HF06iclp]) 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 $\forall$

Two sides of the same coin

Protect the implementation from the outside context.

Do not allow external calls to a predicate that is not exported (*private*).

Protect the outside context from being accessed by the implementation.

Do not allow unrestricted access to the calling context (variables) from inside the implementation.

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 $c \vdash_{\mathcal{C}} x = y \otimes \top$, we get :

Theorem 2 (Code protection [HFS07fsttcs])

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/SWI modules do not offer any 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
```

EMoP modules

EMoP is the implementation by T. Martinez of [HFS07fsttcs]

`http://lifeware.inria.fr/~tmartine/emop/`

```
module 'data.ref.non_backtrackable' {
   new(Initial, Ref) :-
   'kernel':ref_non_backtrackable_new(Initial, X),
   module Ref [Ref, X] {
      get(V) :-
      ...
      set(V) :-
      ...
   }.
}
```

CLP with modules (and closures) as first-class objects, including unification, passing around, environment, etc.

Bonus: functional syntax, modular and redefinable, fully bootstrapped, compiled to native, …

CSR $\Leftrightarrow$ flat-LCC

CSR is the fragment of CHR with only simplification rules:

$$\frac{(H \Leftrightarrow C \mid B)[x/y] \in P \qquad \mathcal{T} \models G_{builtin} \supset \exists x(H = H' \wedge C)}{H' \wedge G \longrightarrow G \wedge H = H' \wedge B}$$

Equivalent to full CHR as far as original operational semantics (and linear logic semantics) are concerned.

[Martinez09chr] shows that CSR can be encoded in LCC:

$$(H \Leftrightarrow C \mid B)^{\dagger} = \forall \vec{y}(C^{\dagger} \otimes H^{\dagger} \Rightarrow \exists \vec{x}.B^{\dagger})$$

where $\vec{x} = fv(B) \setminus fv(H', C)$ and $\vec{y} = fv(H', C)$

The encoding is reciprocal for flat-LCC, i.e., LCC with all asks at top-level.

LCC $\Leftrightarrow$ flat-LCC

Actually LCC itself can be encoded in flat-LCC:

- label each (persistent or not) ask with a new token depending on the free variables it depends on
- move all asks to top-level, adding to their guard the corresponding label
- add tells after each ask for all asks *under* it

Both bisimilarity and semantics preservation hold [Martinez09chr] (Coq proof)

PS: Marelle – Logic Programming for devops

Made HN front page in September 2013.

`http://quietlyamused.org/blog/2013/11/09/marelle-for-devops/`

"At 99designs […] machines should be disposable. This requires the entire setup of a new machine to be automated.

At first I amassed shell scripts of complicated install routines, and whilst these worked they weren't that composable, say when you wanted multiple services on the same machine. Then from Babushka we learned a better way: *test* if something you need's there, *install* it if it's not, then test again to see if you succeeded. This is not hugely different from using make, just more flexible and more fault-tolerant.

Still, Babushka made me uneasy: all this ceremony and complex templating, just to describe a few facts and simple rules?" – Lars Yencken