Closures and Modules within Linear Logic Concurrent Constraint Programming

Rémy Haemmerlé, François Fages and Sylvain Soliman

INRIA Paris-Rocquencourt – France
`FirstName.LastName@inria.fr`

Abstract. There are two somewhat contradictory ways of looking at modules in a given programming language. On the one hand, module systems are largely independent of the particulars of programming languages. On the other hand, the module constructs may interfere with the programming constructs, and may be redundant with the other scope mechanisms of a specific programming language, such as closures for instance. There is therefore a need to unify the programming concepts that are similar, and retain a minimum number of essential constructs to avoid arbitrary programming choices. In this paper, we realize this aim in the framework of linear logic concurrent constraint programming (LCC) languages. We first show how declarations and closures can be internalized as agents in a variant of LCC for which we provide precise operational and logical semantics in linear logic. Then, we show how a complete module system can be represented within LCC, and prove for it a general code protection property. Finally we study the instanciation of this scheme to the implementation of a safe module system for constraint logic programs, and conclude on the generality of this approach to programming languages with logical variables.

1 Introduction

Module systems are an essential feature of programming languages as they facilitate the re-use of existing code and the development of general purpose libraries. There are however two contradictory ways of looking at a module system. On the one hand, a module system is essentially independent of the particulars of a given programming language. "Modular" module systems have thus been designed and indeed adapted to different programming languages, see e.g. [15]. On the other hand, module constructs often interfere with the programming constructs and may be redundant with other scope mechanisms supported by a given programming language, such as closures for instance. There is therefore a need to unify the programming concepts that are similar in order to retain a minimum number of essential constructs and avoid arbitrary programming choices. In this paper, we realize this aim in the framework of linear logic concurrent constraint (LCC) programming languages.

The class of concurrent constraint (CC) programming languages was introduced in [18] as an elegant merge of constraint logic programming (CLP) and concurrent logic programming. In the CC paradigm, CLP goals become concurrent agents communicating through a common store of constraints, each agent being able to post constraints to the store, and to synchronize by asking whether a guard constraint is entailed by the store. Research on the logical semantics of CC languages [6] led to a simple solution in Girard's Linear Logic [8]. Through a straightforward translation of CC agents into intuitionistic LL (ILL) formulas, CC operational transitions indeed correspond to deductions in ILL, and completeness theorems hold for the observation of successes as well as accessible stores [6].

Moreover, the soundness and completeness theorems still hold when considering constraint systems based on Linear Logic instead of classical logic, that constitutes the LCC framework. From a programming point of view, ILL constraint systems are a refinement of classical constraint systems allowing for the non-monotonic evolution of the constraint store, as advocated in [2], through the consumption of Linear Logic tokens by linear implication [6]. In LCC, constraint programming and imperative programming features are thus reconciled in a unified framework, and LCC has been proposed in [9] as a kernel language for developing constraint programming libraries in a modular fashion.

In this paper, we focus on a closure mechanism and a module system that can be naturally internalized in LCC. We first show in Sect. 2, that the linear tokens and the bang operator of LL can be used to internalize CC declarations and procedure calls as respectively constraint posting and constraint asking in LCC. A quite general notion of closure can then be encoded as a banged agent with an environment. The case of an empty environment corresponds to the usual CC declarations. Then in Sect. 3, we develop a complete module system for LCC via a simple syntactical convention for encapsulating procedure declarations and calls. This restriction allows us to prove a general property of code protection by showing that the implementation hiding follows from the usual scope mechanism for variables. This module system is then illustrated in Sect. 4, by its instantiation to constraint logic programming (CLP) languages, and by its relationship to the module system proposed in [10]. Its implementation is discussed there along the lines of its semantics in LCC, and is illustrated with examples of code hiding, closure programming and module parameterization in CLP. Finally, we conclude on the generality of this approach for programming languages with logical variables.

Related Work

Concerning CC languages, the implementation of modules has not been much discussed up to now, being considered as an orthogonal issue. For instance, the MOZART-OZ language [17, 4] contains an *ad-hoc* module system allowing for separate compilation, but presented as an extra logical feature separated from the other programming constructs.

Concerning programming languages developed in Linear Logic using the Logic Programming paradigm, like for instance LO [1], Lolli [13] or Lygon [12], it is worth noticing that persistent asks (which could be represented as implications under a ! in most of these languages) have not been considered, nor the direct encoding of dynamic clause assertions. On the other hand, the banged ask appears in the recent work of [16] on the expressiveness of linearity and persistence in process calculi for security. In LCC, we shall use the full power provided by both persistent and non persistent inputs and outputs.

The internalization of declarations as agents proposed in this paper also goes somehow in the opposite direction to that of definition-based logics, as described for instance in [11]. Here, we represent definitions are represented by banged agents as first-class citizens. This makes it possible to represent closures just by definitions sharing variables with other agents.

2 Declarations as Agents

In this paper, a set of variables is denoted by x, y, z... while a sequence of variables is denoted by $\boldsymbol{x}$, $\boldsymbol{y}$... The set of free variables occurring in a formula A is denoted by $\mathcal{V}(A)$, $A[\boldsymbol{x}\backslash\boldsymbol{t}]$ denotes the formula A in which the free occurrences of variables $\boldsymbol{x}$ have been replaced by terms $\boldsymbol{t}$ (with the usual renaming of bound variables, avoiding variable clashes).

In this section, we give a presentation of LCC languages where the usual CC declarations are replaced by banged ask agents, called *persistent asks*. This construct actually generalizes usual declarations to closures with the environment represented by the free variables in the persistent asks. Before that, we recall the definition of linear logic constraint systems as given in [6].

2.1 Linear Logic Constraint Systems

LCC languages essentially extend CC languages by considering constraint systems based on Girard's Linear Logic [8] instead of classical logic [6]. From a programming point of view, this extension introduces state change and imperative features in constraint languages by allowing a non-monotonic evolution of the store of constraints [2].

Let $\mathcal{T}$ be the set of terms (noted t, s, $\ldots$) formed from a set V of variables and a set Σ_F of function symbols. An *atomic constraint* is a formula built from V, Σ_F and a set Σ_C of relation symbols. The *constraint language* is the least set containing all atomic constraints, closed by multiplicative conjunction ($\otimes$) existential quantification ($\exists$) and exponentiation ($!$).

Definition 1 (LL Constraint System). *A* linear constraint system *is a pair* $(\mathcal{C}, \Vdash_{\mathcal{C}})$ *where* $\mathcal{C}$ *is a constraint language containing* $\mathbf{1}$ *the neutral element of the multiplicative conjunction and* $\Vdash_{\mathcal{C}}$ *is a subset of* $\mathcal{C}\times\mathcal{C}$ *which defines the non-logical axioms of the system. The entailment relation* $\vdash_{\mathcal{C}}$ *is the least subset of* $\mathcal{C}^*\times\mathcal{C}$ *containing* $\Vdash_{\mathcal{C}}$ *and closed by the rules of ILL for* $\mathbf{1}$, $\top$, $!$, $\exists$ *and* $\otimes$.

In this setting, classical constraints are written under a bang !, while linear logic constraints without bang can be consumed by linear implication. In practice, the non classical constraints will be restricted to *linear tokens* which have no axiom, except the general axiom of equality : $l(x) \otimes !(x = y) \vdash l(y)$

The vocabulary of predicate symbols Σ_C is thus partitioned into two sets Σ_D, Σ_L, where Σ_D contains the *classical constraints* with at least *true* ($\mathbf{1}$), *false* ($\mathbf{0}$) and $=$, and Σ_L contains the *linear token predicates*. The constraint languages built from Σ_D and Σ_L are noted $\mathcal{D}$ and $\mathcal{L}$ respectively.

Example 1. A typical LL constraint system is that of a combination of classical constraints, such as Herbrand terms, with linear tokens like $value(x, v)$ that can be added added to and deleted from the store to encode imperative variables and assignment. In the following, linear tokens will also be used to represent procedure calls, by tokens consumed by the procedure definition at the time of its execution.

As no classical constraint but $\mathbf{0}$ can entail a linear token, we have :

Proposition 1. *Let $c \in \mathcal{D}$ and $l \in \mathcal{L}$. If $c \vdash l \otimes \top$ then $c \vdash \mathbf{0}$.*

The set of free variables occurring in the linear tokens of some constraint c is denoted by $\mathcal{V}_l(c)$. Formally, $\mathcal{V}_l(l(\boldsymbol{t})) = \mathcal{V}(\boldsymbol{t})$ if $l \in \Sigma_L$, and $\mathcal{V}_l(l(\boldsymbol{t})) = \emptyset$ if $l \in \Sigma_D$, and this is extended to non-atomic constraints as usual.

2.2 Syntax and Operational Semantics of LCC Agents

Given an LL constraint system $(\mathcal{C}, \Vdash_{\mathcal{C}})$, the syntax of LCC$(\mathcal{C}, \Vdash_{\mathcal{C}})$ agents is defined by the following grammar : $A ::= A \,||\, A \mid \exists x.A \mid c \mid \forall \boldsymbol{x}(c \rightarrow A) \mid \forall \boldsymbol{x}(c \Rightarrow A)$ where c stands for any constraint in $\mathcal{C}$ and $\boldsymbol{x} \subset \mathcal{V}_l(c)$. As usual $||$ stands for parallel composition, the $\exists$ operator hides variables in an agent, and the *tell* agent, written as a constraint, adds that constraint to the store. Two forms of ask agents are considered here : $\forall \boldsymbol{x}(c \rightarrow A)$ for the usual ask, and $\forall \boldsymbol{x}(c \Rightarrow A)$ for the persistent ask that will serve to represent procedure definitions. In both cases we impose $\boldsymbol{x} \subset \mathcal{V}_l(c)$. This restriction limits the binding of variables by pattern matching to the variables occurring in linear tokens, and prevents the possible enumeration of all variables by ask agents.

The choice operator is defined here as an abbreviation as in the classical encoding of the non-deterministic choice in CLP with two clauses with the same head : $A + B = \exists x(choice(x) \,||\, (choice(x) \Rightarrow A) \,||\, (choice(x) \Rightarrow B))$

The operational semantics of LCC with persistent ask is defined similarly to [6] with an equivalence and a transition relation defined over configurations. A *configuration* is a tuple $\langle X; c; \Gamma \rangle$ where X is a set of variables, Γ a multiset of agents and c a constraint, called *store*. $\equiv$ is the least equivalence satisfying the following rule of *parallel composition*: $\langle X; c; A \,||\, B, \Gamma \rangle \equiv \langle X; c; A, B, \Gamma \rangle$

The transition relation $\longrightarrow$ is the least relation satisfying the following rules modulo $\equiv$ (its transitive and reflexive closure is denoted by $\xrightarrow{*}$):

Hiding $$\frac{z \notin X \cup \mathcal{V}(c, \Gamma)}{\langle X;\, c;\, \exists \boldsymbol{z.A}, \Gamma\rangle \longrightarrow \langle X \cup \{z\};\, c;\, \boldsymbol{A}, \Gamma\rangle}$$

Tell $$\langle X;\, c;\, \boldsymbol{d}, \Gamma\rangle \longrightarrow \langle X;\, c \otimes d;\, \Gamma\rangle$$

Ask $$\frac{c \vdash_{\mathcal{C}} \exists Y.(d[\boldsymbol{z}\backslash \boldsymbol{s}] \otimes e) \qquad Y \cap (X \cup \mathcal{V}(A, \Gamma)) = \emptyset}{\langle X;\, c;\, \forall \boldsymbol{z}(\boldsymbol{d} \rightarrow \boldsymbol{A}), \Gamma\rangle \longrightarrow \langle X \cup Y;\, e;\, \boldsymbol{A}[\boldsymbol{z}\backslash \boldsymbol{s}], \Gamma\rangle}$$

Persistent ask $$\frac{c \vdash_{\mathcal{C}} \exists Y.(d[\boldsymbol{z}\backslash \boldsymbol{s}] \otimes e) \qquad Y \cap (X \cup \mathcal{V}(A, \Gamma)) = \emptyset}{\langle X;\, c;\, \forall \boldsymbol{z}(\boldsymbol{d} \Rightarrow \boldsymbol{A}), \Gamma\rangle \longrightarrow \langle X \cup Y;\, e;\, \boldsymbol{A}[\boldsymbol{z}\backslash \boldsymbol{s}], \forall \boldsymbol{z}(\boldsymbol{d} \Rightarrow \boldsymbol{A}), \Gamma\rangle}$$

Definition 2 (Observables). *Let A be an LCC($\mathcal{C}$) agent. We say that a constraint $d \in \mathcal{C}$ is an* accessible constraint *for A if there exists a derivation of the form $\langle \emptyset;\, \mathbf{1};\, A\rangle \xrightarrow{*} \langle X;\, c;\, \Gamma\rangle$ such that $\exists X.c \vdash_{\mathcal{C}} d \otimes \top$. Similarly, d is a* success *for A, if in addition Γ is a multiset of persistent asks , $\exists X.c \vdash_{\mathcal{C}} d$, and $\langle X;\, c;\, \Gamma\rangle \not\longrightarrow$.*

Definition 3 (Operational Semantics).

- $\mathcal{O}^{const}(A)$ *is the set of accessible constraints for the agent A.*
- $\mathcal{O}^{\mathcal{D}const}(A) = \mathcal{O}^{const}(A) \cap \mathcal{D}$ *is the set of accessible* $\mathcal{D}$-constraints *for A.*
- $\mathcal{O}^{succ}(A)$ *is the set of successes for the agent A.*
- $\mathcal{O}^{\mathcal{D}succ}(A) = \mathcal{O}^{succ}(A) \cap \mathcal{D}$ *is the set of* $\mathcal{D}$-successes *for the agent A.*

Example 2. In LCC, the scope mechanism of variables and the persistent ask make it possible to encode *closures*. For instance, the agent $\forall x(apply(c, x) \Rightarrow min(x, minint) \otimes max(x, maxint))$ waits for a token of application of a closure c to a variable x to add new constraints on x. From a functional perspective, C is equivalent to $(\lambda X.min(X, minint) \otimes max(X, maxint))$, and the agent $apply(C, X)$ to $C.X$. This schema for closures makes it possible to define iterators on data structures such as `forall` on lists, passing the closure as an argument as follows (the frist two lines define the iterator and the last one uses it):

$\forall C.forall(C, [\,]) \Rightarrow true \;||$
$\forall H, T, C.forall(C, [H|T]) \Rightarrow apply(C, H) \otimes forall(C, T) \;||$
$\exists C.(\forall X(apply(C, X) \Rightarrow min(X, minint) \otimes max(X, maxint)) \;||\; forall(C, L))$

Example 3. Rewriting rules with constraints such as in the CHR [7] can be easily encoded in LCC. For instance, the three following CHR rules for defining the ordering constraint `=<` assuming the built-in equality constraint `=` :

```
X=<Y <=> X=Y|true.    X=<Y,Y=<X <=> X=Y.    X=<Y,Y=<Z ==> X=<Z.
```

can be represented by the following LCC agent (Note that as in the naive semantics of CHR, the last rule does not terminate) :

$\forall X, Y((X =< Y \otimes X = Y) \Rightarrow \mathbf{1}) \;||$
$\forall X, Y((X =< Y \otimes Y =< X) \Rightarrow X = Y) \;||$
$\forall X, Y, Z((X =< Y \otimes Y =< Z) \Rightarrow (X =< Y \otimes Y =< Z \otimes X =< Z))$

This example illustrates the mixing in ask guards of linear tokens $=<$ with the classical (built-in) constraint $=$.

2.3 Logical Semantics of LCC Agents

In this section, we show how the logical semantics of LCC in ILL [6] extends to persistent asks. The translation of LCC agents into ILL is straightforward :

$$c^\dagger = c \qquad (\exists x.c)^\dagger = \exists x.c^\dagger \qquad (A \mid\mid B)^\dagger = A^\dagger \otimes B^\dagger$$
$$(\forall \boldsymbol{x}(c \rightarrow A))^\dagger = \forall \boldsymbol{x}(c \multimap A^\dagger) \qquad (\forall \boldsymbol{x}(c \Rightarrow A))^\dagger = !\forall \boldsymbol{x}(c^\dagger \multimap A^\dagger)$$

This translation extends to a multiset of agents Γ by $\{A_1, \ldots, A_n\}^\dagger = A_1^\dagger \otimes \cdots \otimes A_n \dagger$, and $\emptyset^\dagger = \mathbf{1}$. The translation of a configuration $\langle X;\, c;\, \Gamma\rangle$ is the formula $\langle X;\, c;\, \Gamma\rangle\dagger = \exists X.(c \otimes \Gamma)$. As in [6], we get:

Theorem 1 (Soundness). *Let $\langle X;\, c;\, \Gamma\rangle$ and $\langle Y;\, d;\, \Delta\rangle$ be two configurations. If $\langle X;\, c;\, \Gamma\rangle \xrightarrow{*} \langle Y;\, d;\, \Delta\rangle$ then $\langle X;\, c;\, \Gamma\rangle^\dagger \vdash_{\mathcal{C}} \langle Y;\, d;\, \Delta\rangle^\dagger$*

Theorem 2 (Completeness). *For any LCC agent A, $\mathcal{O}^{const}(A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{\mathcal{C}} c \otimes \top\}$, $\mathcal{O}^{\mathcal{D}const}(A) = \{d \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}} d \otimes \top\}$.*

Because LCC declarations are represented here with persistent asks using the bang operator, the logical characterization of successes requires persistent asks to have a linear token in their guard :

Definition 4 ($\mathcal{L}$-persistent). *Let $\mathcal{C}$ be a constraint system partitioned into classical constraints $\mathcal{D}$ and linear tokens $\mathcal{L}$. An agent is $\mathcal{L}$-persistent if the guards in its persistent asks all contain tokens in $\mathcal{L}$.*

Theorem 3 (Completeness on $\mathcal{D}$-successes). *For any $\mathcal{L}$-persistent LCC($\mathcal{C}$) agent A for which $\mathbf{0}$ is not an accessible constraint we have $\mathcal{O}^{\mathcal{D}succ}(A) = \{d \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}} d\}$.*

3 Modules as Agents

3.1 Syntactical Conventions

The declaration and closure mechanism provided by the persistent ask in LCC can be used to build a complete module system within LCC. In this approach, a module is named by a variable and the scope of module declarations thus depends on the scope of these variables. It is worth noting that for the issue of separate compilation not considered here, modules should also be named by constants making them visible by separate modules. That will be used in the next section.

We use the syntactical convention $x\{A\}$ to denote the agent A in module x. Similarly, telling a token constraint l of module x is denoted by $x\!:\!l$, while classical constraints are not localized. With these conventions, the syntax of modular LCC (MLCC) agents is the following: $A ::= x\{A\} \mid x\!:\!l \mid d \mid A || A \mid \exists x.A \mid \forall \boldsymbol{x}(c \rightarrow A) \mid \forall \boldsymbol{x}(c \Rightarrow A)$ where l stands for a linear token constraint, d stands for a classical constraint and c stands for an arbitrary constraint.

Now, MLCC agents are translated into LCC agents over a modified constraint system, noted $\dot{\mathcal{C}}$, in which an extra argument is added to every linear token. The resulting LCC agents enjoy some sort of code protection as shown in next section.

Definition 5 (Translation in LCC). *For any variable x referencing a module, the translation $()^x$ of MLCC($\mathcal{C}$) agents to LCC($\dot{\mathcal{C}}$) agents is defined recursively by:*

$$d(\boldsymbol{t})^x = d(\boldsymbol{t}) \qquad l(\boldsymbol{t})^x = \dot{l}(x, \boldsymbol{t}) \qquad (c \otimes c')^x = c^x \otimes c'^x \qquad (\exists \boldsymbol{y}.c)^x = \exists \boldsymbol{y}.c^x$$
$$(\exists \boldsymbol{y}.A)^x = \exists \boldsymbol{y}.A^x \qquad (y\{A\})^x = A^y \qquad (y\!:\!l)^x = l^y \qquad (A \mid\mid B)^x = A^x \mid\mid B^x$$

$(!c)^x = !c^x \qquad (\forall \boldsymbol{y}(c \to A))^x = \forall \boldsymbol{y}(c^x \to A^x) \qquad (\forall \boldsymbol{y}(c \Rightarrow A))^x = \forall \boldsymbol{y}(c^x \Rightarrow A^x)$
where $\boldsymbol{y} \cap \mathcal{V}(x) = \emptyset$, $d \in \Sigma_D$, $l \in \Sigma_L$, $c \in \mathcal{C}$ *and* $c' \in \mathcal{C}$.

An LCC($\dot{\mathcal{C}}$*) agent A is* modular *if it is the translation of an MLCC(*$\mathcal{C}$*) agent i.e. there exists an MLCC(*$\mathcal{C}$*) agent B and a variable x such that* $A = B^x$. *An LCC(*$\dot{\mathcal{C}}$*) configuration is* modular *if all its agents are modular.*

Example 4. With these conventions, a module for lists can be defined with internal anonymous modules for hiding the implementation of predicates, such as the *reverse* predicate with a ternary implementation using an accumulator :

$$\begin{aligned}&List\{\exists I.\ (\ \forall X, Y.reverse(X, Y) \Rightarrow I : reverse(X, [], Y) \ ||\\ &\qquad I\ \{\ \forall X, Y.reverse([], X, Y) \Rightarrow !(X = Y)\ ||\\ &\qquad\quad \forall X, Y, Z, T.reverse([X|Y], Z, T) \Rightarrow reverse(Y, [X|Z], T).)\}\}\end{aligned}$$

For the sake of readability, in the following section, constraints of $\dot{\mathcal{C}}$ and agents of LCC($\dot{\mathcal{C}}$) will be denoted respectively by $\dot{c}, \dot{d}, \dot{e} \ldots$ and by $\dot{A}, \dot{B} \ldots$, whereas constraints of $\mathcal{C}$ and agents of MLCC($\mathcal{C}$) will be denoted respectively by $c, d, e \ldots$ and by $A, B \ldots$. Moreover, note that if κ is a modular configuration and $\kappa \xrightarrow{*} \kappa'$ then κ' is modular.

3.2 Code Protection

MLCC programs enjoy a general property of code protection provided that the constraint system does not allow to make arbitrary variables equal. This is enforced by assuming that $\{x, y\} \subset \mathcal{V}(c)$ whenever $c \Vdash_{\mathcal{C}} x = y \otimes \top$ for any distinct variables x and y.

Definition 6. *Let* $\langle X;\ \dot{c};\ \Delta, \dot{B}, \dot{l}\rangle$ *be a modular configuration. The transitions from* $\dot{B}$ *are* independent *from the linear tell agent* $\dot{l}$ *if for any derivation that first reduces tell* $\dot{l}$ *then B, i.e. of the form :*

$$\langle X;\ \dot{c};\ \Delta, \dot{B}, \dot{l}\rangle \longrightarrow \langle X;\ \dot{c} \otimes \dot{l};\ \Delta, \dot{B}\rangle \longrightarrow \langle Y;\ \dot{d}';\ \Delta, \dot{B}'\rangle$$

there exists a derivation that first reduces B then l of the form:

$$\langle X;\ \dot{c};\ \Delta, \dot{B}, \dot{l}\rangle \longrightarrow \langle Y';\ \dot{e};\ \Delta, \dot{B}', \dot{l}\rangle \longrightarrow \langle Y;\ \dot{c}' \otimes \dot{l};\ \Delta, \dot{B}'\rangle \qquad \text{with } \dot{c}' \otimes \dot{l} \dashv\vdash_{\dot{\mathcal{C}}} \dot{d}' \quad .$$

Definition 7 (Code Protection). *An agent* $\dot{A}$ *is* protected *in a modular agent* $C[\dot{A}]$ *if the transitions from* $\dot{B}$ *are independent from* $\dot{l}$ *in any configuration* $\langle X;\ \dot{c};\ \Delta, \dot{B}, \dot{l}\rangle$ *such that* $\langle \emptyset;\ \mathbf{1};\ C[\dot{A}], \Gamma\rangle \xrightarrow{*} \langle X;\ \dot{c};\ \Delta, \dot{B}, \dot{l}\rangle$, $\dot{B}$ *derives from* $\dot{A}$ *and* $\dot{l}$ *derives from* Γ.

Theorem 4. *Let A and B be two MLCC(*$\mathcal{C}$*) 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* $y \notin \mathcal{V}(l)$, *then* $(A)^y$ *is protected in* $(\exists y.(y\{A\} \mid\mid B))^x$ *for any variable x.*

The proof of this theorem relies on general properties on the scope of variables, and on technical properties of constraint decomposability and variable accessibility. The intuition behind decomposability is that linear tokens can be separated from the rest of the constraint without making it logically weaker.

Definition 8 (Decomposable constraint). *A constraint is in* separated form *if it is of the form* $d \otimes \dot{l}_1 \otimes \cdots \otimes \dot{l}_k$ *where d is a classical constraint and the* $\dot{l}_i$*'s are atomic linear token constraints. A constraint is in* decomposed form *if it is of the form* $\exists Y.\dot{d}$ *where* $\dot{d}$ *is in separated form. A constraint is* decomposable *(resp.* separable*) if it is equivalent to a decomposed (resp. separated) form.*

Lemma 1. *Let* Γ *be a multiset of consistent constraints in decomposed form,* $\dot{c} \in \dot{\mathcal{C}}$ *a constraint, and Y a set of variables. If* $\Gamma \vdash_{\dot{\mathcal{C}}} \exists Y.(\dot{c} \otimes \top)$ *then* $\dot{c}$ *is decomposable.*

Proposition 2. *Let* $\langle X;\, \dot{c};\, \Gamma\rangle \xrightarrow{*} \langle Y;\, \dot{d};\, \Delta\rangle$ *be a derivation between two modular configurations. If* $\dot{c}$ *is consistent and decomposable then* $\dot{d}$ *is decomposable.*

Let $\dot{c} \in \dot{\mathcal{C}}$ be a separable constraint and X a set of variables. We define the set of variables *accessible by unification* in $\dot{c}$ from X as :
$$\mathcal{A}^u_{\dot{c}}(X) = X \cup \{x \in \mathcal{V}(\dot{c})|\; d \in \mathcal{D},\; y \in \mathcal{V}(d),\; \mathcal{V}(\dot{c}) \cap \mathcal{V}(d) \subset X,\\ \dot{c} \otimes d \vdash_{\dot{\mathcal{C}}} x{=}y \otimes \top \text{ and } \Gamma, d \not\vdash_{\dot{\mathcal{C}}} \mathbf{0}\}.$$
The set of variables *accessible by substitution* in $\dot{c}$ from X is :
$$\mathcal{A}^s_{\dot{c}}(X) = X \cup \left\{x \in \mathcal{V}(\boldsymbol{t}) \,\middle|\, \dot{l} \in \Sigma m \quad y \in X \quad \dot{c} \vdash_{\dot{\mathcal{C}}} \dot{l}(y, \boldsymbol{t}) \otimes \top \quad \dot{c} \not\vdash_{\dot{\mathcal{C}}} \mathbf{0}\right\}$$
The set of *directly accessible* variables in $\dot{c}$ from X is $\mathcal{A}^1_{\dot{c}}(X) {=} \mathcal{A}^s_{\dot{c}}(X) \cup \mathcal{A}^u_{\dot{c}}(X)$.

Proposition 3. *For any consistent separable constraint* $\dot{c}$, $\mathcal{A}^1_{\dot{c}}$ *is extensive, monotone and bound.*

This proposition allows us to define the set of *accessible* variables in $\dot{c}$ from X, noted $\mathcal{A}_{\dot{c}}(X)$, as the least fix point of $\mathcal{A}^1_{\dot{c}}$ containing X. The set of *accessible* variables in a decomposable constraint $\dot{d}$ is $\mathcal{A}_{\dot{c}}(X) \setminus Y$ where Y is a set of variables and $\dot{c}$ is a separable constraint such that $\dot{d} \vdash_{\dot{\mathcal{C}}} \exists Y.\dot{c}$ and without loss of generality $Y \cap X = \emptyset$.

Lemma 2. *Let* $\dot{c}$ *and* $\dot{d}$ *be two consistent decomposable constraints of* $\dot{\mathcal{C}}$ *and* X *an arbitrary set of variables. If* $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d} \otimes \top$ *then* $\mathcal{A}_{\dot{c}}(X) \supset \mathcal{A}_{\dot{d}}(X)$.

Proposition 4. *Let* $\langle X;\, \dot{c};\, \Gamma, \Delta\rangle \xrightarrow{*} \langle Y;\, \dot{d};\, \Gamma', \Delta'\rangle$ *be a derivation between two consistent configurations such that* Γ' *be the reduced of* Γ, *and* $\dot{c}$ *is decomposable. If* $x \in \mathcal{V}(X, \Delta)$ *and* $x \notin \mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma))$ *then* $x \notin \mathcal{A}_{\dot{d}}(\mathcal{V}(\Gamma'))$.

4 Implementation as a Module System for CLP

The MLCC scheme presented above instantiates into a powerful module system for Constraint Logic Programming languages, called mCLP. This module system is an extension including dynamic modules of the module system proposed for CLP in [10]. It is provided here with a logical semantics in linear logic, and with an implementation with closures in the line of its semantics in LCC. A prototype implementation of mCLP is available for download at `http://contraintes.inria.fr/~haemmerl/pub/mclp.tgz`.

4.1 mCLP Syntactical Conventions

We shall adopt for mCLP a pragmatic syntax close to that of classical CLP systems. The `typewriter` font is used for programs, where, as in classical Prolog programs, the identifiers beginning with a capital letter represent variables. The syntax defined by the following grammar distinguishes declarations from goals as usual:

$$\mathtt{G} ::= \mathtt{module(T,E)\{D\}} \mid \mathtt{T} : \mathtt{p(S_1,\ldots,S_n)} \mid \mathtt{p(S_1,\ldots,S_n)} \mid \mathtt{c(S_1,\ldots,S_n)} \mid \mathtt{G,G} \mid \mathtt{G;G}$$
$$\mathtt{D} ::= \mathtt{p(S_1,\ldots,S_n)} : -\mathtt{G.D} \mid \mathtt{p(S_1,\ldots,S_n).D} \mid : -\mathtt{G.D} \mid \epsilon$$

where `T` is a term, `E` a list of variables, $\mathtt{S_1},\ldots,\mathtt{S_n}$ a sequence of terms, `c` a constraint of $\mathcal{C}$ and `p` a predicate construct using the alphabet Σ_L.

An mCLP declaration is either a clause, a fact or a goal of the form `:- G.` executed at the initialization of the module. Besides the usual conjunction, disjunction and constraint posting goals, the goal `module(T, E){D}` denotes the *instantiation* of a module `T` with the *implementation* `D` and the *environment* `E`. This environment is simply a list of *global variables* whose scope is the entire module clauses. If `T` is a free variable, the resulting module is *anonymous*, whereas if `T` is an atom (or a compound term), it is a *named* module, as proved useful for separate compilation. The goal $\mathtt{T{:}p(S_1,\ \ldots,\ S_n)}$ denotes the *external call* of the predicate `p/n` defined in the module `T`, which is distinguished from the *local call*, noted $\mathtt{p(S_1,\ \ldots,\ S_n)}$, of the predicate `p/n` defined in the current module.

4.2 Interpretation and Compilation

Classical clauses are interpreted by *persistent asks* waiting for the linear token that represents the procedure call. The module environment provides a new feature allowing for global variables in a module. Formally, the interpretation of mCLP goals and declaration in MLCC is defined by $[\![\mathtt{G}]\!]^{\mathtt{T}}$ and $[\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$ where `T` is the current module and `E` the current environment:

$$[\![\mathtt{G_1,G_2}]\!]^{\mathtt{T}} = [\![\mathtt{G_1}]\!]^{\mathtt{T}} \mid\mid [\![\mathtt{G_2}]\!]^{\mathtt{T}} \qquad [\![\mathtt{P}]\!]^{\mathtt{T}} = \mathtt{T{:}P} \qquad [\![\mathtt{S{:}P}]\!]^{\mathtt{T}} = \mathtt{S{:}P}$$
$$[\![\mathtt{G_1;G_2}]\!]^{\mathtt{T}} = [\![\mathtt{G_1}]\!]^{\mathtt{T}} + [\![\mathtt{G_2}]\!]^{\mathtt{T}} \qquad [\![\mathtt{C}]\!]^{\mathtt{T}} = \mathtt{T{:}(!C)} \qquad [\![\mathtt{module(S,E)\{D\}}]\!]^{\mathtt{T}} = \mathtt{S}\{[\![\mathtt{D}]\!]^{\mathtt{S}}_{\mathtt{E}}\}$$
$$[\![\,\mathtt{{:}\text{-}\,G.D}]\!]^{\mathtt{T}}_{\mathtt{E}} = \exists\overline{\mathtt{Y}}[\![\mathtt{G}]\!]^{\mathtt{S}} \mid\mid [\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}} \qquad [\![\mathtt{p(t).D}]\!]^{\mathtt{T}}_{\mathtt{E}} = \forall\mathtt{X}(\mathtt{p(X)} \Rightarrow \exists\overline{\mathtt{Y}}[\![\mathtt{X{=}t}]\!]^{\mathtt{S}}) \mid\mid [\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$$
$$[\![\mathtt{p(t)\,{:}\text{-}\,G.D}]\!]^{\mathtt{T}}_{\mathtt{E}} = \forall\mathtt{X}(\mathtt{p(X)} \Rightarrow \exists\overline{\mathtt{Y}}[\![\mathtt{X = t,G}]\!]^{\mathtt{S}}) \mid\mid [\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$$

where `X` is a set of fresh variables and $\overline{\mathtt{Y}} = \mathcal{V}(\mathtt{t},\mathtt{G}) \setminus \mathtt{E}$.

Let $\Vdash_{\mathcal{C}^\circ}$ be the translation of the constraint system $\mathcal{C}$ into linear logic (using for example the well know Girard's translation classical logical into LL [8]). The constraint system $(\mathcal{CP}, \Vdash_{\mathcal{CP}})$ corresponding to this translation is defined such that $\Vdash_{\mathcal{CP}}$ is the smallest set respecting the following conditions (1) if $(\mathtt{C} \Vdash_{\mathcal{C}^\circ} \mathtt{C})$ then $(\mathtt{C} \Vdash_{\mathcal{CP}} \mathtt{C})$ and (2) for any predicate symbol `p` $(\mathtt{p(X)}, \mathtt{X{=}Y} \Vdash_{\mathcal{CP}} \mathtt{p(Y)})$.

Notice that all the $[\![A]\!]^{\mathtt{T}}_{\mathtt{E}}$ are $\mathcal{L}$-persistent (see Def. 4), therefore all results of previous Section can be applied to mCLP programs.

In addition to a first order logical semantics, this translation provides a way to compile mCLP using classical Prolog compilation techniques. Typically a module is referenced by a special variable to which module environment and module procedures are attached as attributes [14]. A mCLP predicate is then implemented by a Prolog predicate with an extra-argument, inherit from logical semantics of persistent (c.f. Def. 5) storing the current module variable.

4.3 Global Variables

Module environments introduce *global* variables, i.e. variables shared among the different clauses of the module. This construct can be used for instance to avoid passing an argument to numerous module predicates. However, these variables are still usual, backtrackable, logic variables.

The following code illustrates the use of a global variable `Depth` to implement a Prolog meta-interpreter with a fair search strategy proceeding by iterative deepening [19]. The predicate *clause* looks for clause definitions [5]; the predicate `for(I, Begin, End)` produces a choice point where `I` will be assigned any of the integer values between `Begin` and `End` (see for instance [3]).

Example 5. (Iterative Deepening):

```
:-module(iter_deep, [Depth]){
      solve(G):- for(Depth,1,1000), iter_deep(G,0).
      iter_deep(_,I) :- I >= Depth, !, fail.
      iter_deep(((A,B)),I) :- !, iter_deep(A,I), iter_deep(B,I).
      iter_deep(A,_) :- clause((A:-true)), !.
      iter_deep(A,I) :- clause((A:-B)), J is I+1, iter_deep(B,J).   }.
```

4.4 Code Hiding

As above, one can use an environment to make a variable *global* to a module, but this time, this variable will be used to keep an anonymous inside module hidden from the outside. Since the *name* of the inside module is this variable, only accessible to the clauses inside the module definition, the corresponding implementation is protected from the clauses outside the external module.

This is illustrated in the following program that provides the `sort` predicate and hides the implementation `quicksort` predicate.

Example 6. (Quicksort):

```
:- module(sort, [Impl]){
      sort(Lst,SrtdLst):- Impl:quicksort(Lst,SrtdLst).
      :- module(Impl,[]){
            quicksort([],[]).
            quicksort([X|Tl],Srtd) :- split(X,Tl,Smll,Bg),
              quicksort(Smll,SrtdSmll), quicksort(Bg,SrtdBg),
              list:append(SrtdSmll,[X|SrtdBg],Srtd).
            split(X,[],[],[]).
            split(X,[Y|Tl],[Y|Smll],Bg) :- X<Y,!,split(X,Tail,Small,Big).
            split(X,[Y|Tl],Smll,[Y|Bg]) :- split(X,Tl,Smll,Bg).   }. }.
```

The code protection property 3.2 ensures that no call to the `quicksort` predicate is possible outside the `sort` predicate. The execution of the following goal prints on screen the sorted list `[2/7,1/2,2/3,1,4/3,5]`.

```
? L=[1, 2/3, 5, 4/3, 1/2, 2/7], sort:sort(L, L1), print(L1), nl.
```

4.5 Closures

The classical notion of *closure* can be recovered through the definition of modules with a predicate `apply/1` waiting for the argument to apply the persistant ask (corresponding to the clauses of `apply/1`).

This makes it possible to define iterators on data structures such as `forall` on lists, passing the closure as an argument as follows:

Example 7. :

```
:- module(iterator, []){
      forall([], _).
      forall([H|T], C) :- C:apply(H), forall(T, C).  }.
```

The usual `domain/3` (or `fd_domain/3`) built-in predicate of finite domain constraint solvers, can be implemented using the list iterator on its arguments:

```
fd_domain(Vars,Min,Max):-module(Cl,[Min,Max]){apply(X):-Min=<X,X=<Max.},
  (list(Vars)->iterator:forall(Vars, Cl) ; var(Vars)->Cl:apply(Vars)).
```

4.6 Module Parameterization

Parameterized modules greatly enhance the programmer capabilities to re-use code by making its module implementation depend on other modules. Combining the idea of using the environment to parameterize a closure, and the code hiding features demonstrated above, one can obtain a module with a hidden implementation, parameterized from outside. The following example shows how to parameterize the previous *sort* module by creating a `generic_sort/2` predicate that dynamically creates a sorting module (its first argument) using the comparison predicate given as second argument.

Example 8. (Parameterized quicksort):

```
:- module(sort, []){
     generic_sort(Sort,Order):- module(Sort,[Order, Impl]){
        sort(List,SortedList):-Impl:qsort(List,SortedList).
        :-module(Impl, [Order]){
           qsort([],[]).
           qsort([X|T],Srtd):-split(X,T,Smll Bg),qsort(Smll,SrtdSmll),
             qsort(Bg,SrtdBg),list:append(SrtdSmll,[X|SrtdBg],Srtd).
           split(X,[],[],[]).
           split(X,[Y|T],[Y|Smll],Bg):-Order:(X >= Y),!,
             split(X,T,Smll, Bg).
           split(X,[Y|T],Smll,[Y|Bg]):-split(X,T,Smll,Bg).  }. }.}.
```

Let `math` be a module defining the ordering predicate `>=` over numbers, and `term` a module defining the ordering predicate `@>=` over terms. The execution of the following goal prints the lists `[2/7,1/2,2/3,1,4/3,5]` and `[1,5,1/2,2/3,2/7,4/3]` which shows the parameterized use of the module `sort`.

```
?- L=[1, 2/3, 5, 4/3, 1/2, 2/7],
    sort:factory(Sort1, math), Sort1:sort(L, L1), print(L1), nl,
    module(OrderLex, []) X >= Y:- term:(X @>= Y) ,
    sort:factory(Sort2, OrderLex), Sort2:sort(L, L2) print(L2), nl.
```

5 Conclusion

We have shown that a powerful module system for linear concurrent constraint programming (LCC) languages can be internalized into LCC, by representing declarations by persistent asks, referencing modules by variables and thus benefiting from implementation hiding through the usual hiding operator for variables. We have presented the operational semantics of MLCC programs, showing a code protection property, and proving the equivalence with the logical semantics in linear logic for the observation of stores and successes.

These results have been illustrated with an instantiation of the MLCC scheme to constraint logic programs, leading to a simple yet powerful module system similar to the one proposed in [10], supporting code hiding, closures and module parameterization, and provided here with a simple logical semantics in linear logic. Another interesting use is the boostrapping of a complete implementation of LCC that is currently under development [9].

We believe that this approach to internalizing a module system within a programming language is of a quite general scope for programming languages with logical variables, as well as its implementation with a closure mechanism.

References

1. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.
2. E. Best, F. S. de Boer, and C. Palamidessi. Concurrent constraint programming with information removal. In *Proceedings of Coordination*, LNCS. Springer-Verlag, 1997.
3. D. Diaz. *GNU Prolog user's manual*, 1999–2003.
4. D. Duchier, L. Kornstaedt, C. Schulte, and G. Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. draft, 1998.
5. P. D. A. Ed-Dbali and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, 1996.
6. F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Infor. and Comput.*, 165(1):14–41, Feb. 2001.
7. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, Oct. 1998.
8. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.
9. R. Haemmerlé. SiLCC is linear concurrent constraint programming (doctoral consortium). In M. Gabbrielli and G. Gupta, editors, *Proceedings of ICLP 2005*, volume 3668 of *LNCS*, pages 448–449. Springer-Verlag, 2005.
10. R. Haemmerlé and F. Fages. Modules for Prolog revisited. In *Proceedings of ICLP 2006*, volume 4079 in LNCS, pages 41–55. Springer-Verlag, 2006.
11. L. Hallnäs. A proof-theoretic approach to logic programming. ii. programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, Oct. 1991.
12. J. Harland, D. J. Pym, and M. Winikoff. Programming in lygon: An overview. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, Munich*, pages 391–405, July 1996.
13. J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
14. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. TR-92-23, Österreichisches Forschungsinstitut für AI, Wien, 1992.
15. X. Leroy. A modular module system. *J. of Func. Prog.*, 10(3):269–303, 2000.
16. C. Palamidessi, V. A. Saraswat, F. D. Valencia, and B. Victor. On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In *Proc. of the 21th Annual IEEE Symposium on Logic In Computer Science*, pages 59–68, 2006.
17. P. V. Roy et al. Logic programming in the context of multiparadigm programming: the Oz experience. *TPLP*, 3(6):715–763, Nov. 2003.
18. V. A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993.
19. M. E. Stickel. A prolog technology theorem prover: implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 44:353–380, 1988.