On Internalizing Modules as Agents in Concurrent Constraint Programming

Remy Haemmerle, Francois Fages, Sylvain Soliman

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On Internalizing Modules as Agents in Concurrent Constraint Programming

Rémy Haemmerlé — François Fages — Sylvain Soliman

N° 5981

Septembre 2006

Thème SYM

ISSN 0249-6399

On Internalizing Modules as Agents in Concurrent Constraint Programming

Rémy Haemmerlé , François Fages , Sylvain Soliman

Thème SYM — Systèmes symboliques
Projet Contraintes

Rapport de recherche n° 5981 — Septembre 2006 — 30 pages

Abstract: 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 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, and several examples of module systems have indeed been adapted to different programming languages. On the other hand, the module constructs often interfere with the programming constructs, and may be redundant with other scope mechanisms of programming languages, such as closures for instance. There is therefore a need to unify the programming concepts and constructs 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 LCC. We then present a modular version of LCC (MLCC), where modules are referenced by variables and where implementation hiding is obtained with the usual hiding operator for variables. We develop the logical semantics of MLCC in linear logic, and show the completeness of the operational semantics for the observation of successes and accessible stores. Finally we discuss a complete module system for constraint logic programming, derived from the MLCC scheme.

Key-words: Modules, agents, closures, code protection

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

Sur l'internalisation des modules en tant qu'agents dans la programmation concurrente avec contraintes

Résumé : Les systèmes de modules sont un trait essentiel des langages de programmation, car ils facilitent la réutilisation du code préexistant et le développement de bibliothèques génériques. Il y a cependant deux façons quelque peu contradictoires de considérer les modules dans un langage de programmation. D'un coté, les systèmes de modules sont largement indépendants des particularités d'un langage de programmation, et plusieurs exemples de systèmes de modules ont en effet été adaptés à différents langages de programmation. D'un autre coté, les constructions de modules interfèrent souvent avec les opérateurs de programmation, et peuvent être redondants avec d'autres mécanismes de liaison, tels que les fermetures par exemple. Il y a donc un besoin pour unifier les concepts et opérateurs de programmation qui sont similaires, et retenir un nombre minimal de constructions essentielles afin d'éviter des choix arbitraires de programmation. Dans cet article, nous réalisons cet objectif dans le contexte des langages de programmation concurrente avec contraintes en logique linéaire (LCC). Nous montrons d'abord comment les déclarations et les fermetures peuvent être internalisées comme des agents LCC, puis nous présentons une version modulaire de LCC (MLCC) où les modules sont référencés par des variables logiques, et où le masquage de l'implantation est obtenu à l'aide de l'opérateur usuel de masquage des variables. Nous développons la sémantique logique de MLCC, et démontrons la correction et la complétude de la sémantique opérationnelle pour l'observation des stores accessibles et des succès. Finalement nous présentons un système de modules pour la programmation logique avec contraintes dérivé de MLCC.

Mots-clés : Modules, agents, fermetures, protection du code

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 [13]. 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 and constructs that are similar in order to retain a minimum number of essential constructs and avoid arbitrary programming choices.

In this paper, we study a complete module system for linear concurrent constraint (LCC) programming languages and show how modules and closures are unified as a particular kind of LCC agents in this framework.

Linear concurrent constraint programming

The class of Concurrent Constraint (CC) programming languages has been introduced in [17] as an elegant merge of constraint logic programming (CLP) and concurrent logic programming. In the CC paradigm, CLP goals are 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. Both theoretical reasons concerning the logical semantics of CC languages [6, 18], and practical reasons concerning the need for a non-monotonic evolution of the store [2], led to a natural extension of CC languages with constraint systems based on Linear Logic (LL) [8], called Linear Concurrent Constraint (LCC) programming. By interpreting CC agents by LL formulae, it is indeed possible to identify CC operational transitions with LL deductions, and obtain completeness theorems for the observation of the set of accessible stores, as well as for the set of success stores [6]. This means that Linear Logic is the logic of CC agents. Moreover, the theorems still hold when considering constraint systems based on Linear Logic instead of classical logic. From a programming point of view, LL constraint systems are a refinement of classical constraint systems allowing for state change and non-monotonic evolution of the constraint store, through the consumption of linear logic tokens by linear implication [6, 2]. This makes it possible to encode imperative features in LCC and combine them with constraint programming.

In this paper, we show that the linear tokens and the bang operator of LCC can be used to internalize CC declarations and procedure calls as constraint posting and asking. A quite general notion of *closure* can then be encoded as a banged agent with an environment, declarations corresponding to the case of an empty environment. These results are then used to define the operational semantics of modular LCC (MLCC) languages, where modules are variables and where implementation hiding is realized with the usual hiding operator for variables.

In Section 4 we provide an equivalent logical semantics where modular LCC agents are interpreted by linear logic formulae, and prove completeness theorems for the observation of success and accessible stores.

Then in Section 5, we derive from the MLCC scheme a powerful module system for constraint logic programming. We illustrate the expressiveness of this module system with examples of code hiding, closure programming and module parameterization in CLP, and discuss its implementation along the lines of its semantics in LCC.

Finally, we conclude on these results and on their generality.

Related Work

The proposed internalization of declarations as agents goes somewhat in the opposite direction to that of definition-based logics, as described for instance in [10]. Here we make definitions first-order objects, which allows us to manipulate them easily, and to generalize them to closures.

There has been several programming languages developed in Linear Logic using the Logic Programming paradigm, like for instance LO [1], Lolli [12] or Lygon [11]. However, for efficiency reasons in these languages, there is no equivalent for the persistent asks (which would be implications under a ! in most of these languages) and thus no direct encoding of dynamic clause assertions as we will do in Sect. 2.4.3. The banged ask appears in the recent work of [14] on the expressiveness of linearity and persistence in process calculi for security.

Concerning CC languages, the implementation of modules has not been much discussed, being considered as an orthogonal issue. For instance, the MOZART-OZ language [15, 4] contains an *ad-hoc* module system allowing for separate compilation. Here we provide a natural integration of module and programming concepts with the limited set of LCC programming constructs.

2 LCC with Declaration Agents

In this section, we give a presentation of the LCC languages where declarations are replaced by banged asks, which we will call *persistent ask*. This new construct actually generalizes declarations into persistent asks by allowing variables to remain free in a persistent ask and represent the environment.

In this paper, a set of variables is denoted by $\overline{x}$ or $\overline{y}$. The set of free variables occurring in a formula A is denoted by $\mathrm{fv}(A)$, a sequence of variables is denoted by $\vec{x}$, $A[\vec{x}\backslash\vec{t}]$ denotes the formula A in which the free occurrences of variables $\vec{x}$ have been replaced by terms $\vec{t}$ (with the usual renaming of bound variables, avoiding variable clashes). For a transition relation $\longrightarrow$, $\longrightarrow^\star$ denotes the transitive and reflexive closure of $\longrightarrow$. The `typewriter` font is used for programs, where, as in classical Prolog programs, the identifiers beginning by a capital letter represent variables.

2.1 Linear Logic Constraint Systems

The class of LCC languages essentially extends CC languages by considering constraint systems based on Linear Logic [8] instead of classical logic. From a programming point of view, this extension introduces state change and imperative features in constraint languages. We recall here the usual definitions of a Linear Logic constraint system (see for instance [6]).

Definition 2.1 (Constraint Language) *An* atomic constraint *is a formula built from a set V of variables,a set Σ_F of function symbols and a set $\Sigma_{\mathcal{C}}$ of relation symbols, which does not contain $\top$, the neutral elements of additive linear conjunctions. The* constraint language *is the least set containing all atomic constraints, marked or not by the unary exponential connective* ! *(called also "bang") and closed by multiplicative conjunction ($\otimes$) and existential quantification ($\exists$).*

Definition 2.2 (Constraint System) *A* linear constraint system *is a pair $(\mathcal{C}, \Vdash_{\mathcal{C}})$ where:*

- *$\mathcal{C}$ is a constraint language.*
- *$\Vdash_{\mathcal{C}}$ is a subset of $\mathcal{C} \times \mathcal{C}$ which defines the non-logical axioms of the constraint system. We suppose that for all free variables occurring in c have a free occurrence in $c_1, \ldots, c_n$.*

We will note $\vdash_{\mathcal{C}}$ the least subset of $\mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the of intuitionistic linear logic, noted in the following ILL (see appendix A for the complete sequent calculus).

Let $\mathcal{C}$ be a constraint system. In the following, $\mathcal{T}$ will be the language of terms (noted t, s, ...) formed from V and Σ_F.

2.2 Syntax of LCC($\mathcal{C}$)

The syntax of LCC($\mathcal{C}$) is presented here without declarations, only agents with two forms of ask agents.

Definition 2.3 *The syntax of LCC($\mathcal{C}$)* agents *is given by the following grammar:*
$A ::= A || A \mid \exists x.A \mid c \mid \forall \vec{x}(c \rightarrow A) \mid \forall \vec{x}(c \Rightarrow A)$

As usual || stands for parallel composition, the *tell* agent adds a constraint to the store, $\exists$ hides variables in an agent and $\rightarrow$ stands for ask. The new construct $\Rightarrow$ represents an *ask* operator, called *persistent ask*, that always remains active.

Note that we do not provide an explicit choice operator, since the local choice operator can easily be encoded with linear tokens and ask as follows:

$$A + B = \exists x(choice(x) \mid\mid choice(x) \Rightarrow A \mid\mid choice(x) \Rightarrow B)$$

This encoding corresponds to the classical encoding of + in CLP as two clauses with the same head.

2.3 Operational Semantics

As usual, the operational semantics of LCC is defined here with a structural congruence and a transition relation defined over configurations.

Definition 2.4 (Configuration) *A* configuration *is a tuple* $\langle \overline{x};\ c;\ \Gamma\rangle$ *where* $\overline{x}$ *is a multi-set of variables,* Γ *a multi-set of agents and* c *a constraint, called* store.

Definition 2.5 *The structural congruence* $\equiv$ *is the least congruence satisfying the following rule of* parallel composition*:*

$$\langle \overline{x};\ c;\ A||B, \Gamma\rangle \equiv \langle \overline{x};\ c;\ A, B, \Gamma\rangle$$

Definition 2.6 *The transition relation* $\longrightarrow$ *is the least relation satisfying the rules of the table 1.*

Equivalence	$\dfrac{\langle \overline{x};\ c;\ \Gamma\rangle \equiv \langle \overline{x};\ c';\ \Gamma'\rangle \longrightarrow \langle \overline{y};\ d';\ \Delta'\rangle \equiv \langle \overline{y};\ d;\ \Delta\rangle}{\langle \overline{x};\ c;\ \Gamma\rangle \longrightarrow \langle \overline{y};\ d;\ \Delta\rangle}$
Tell	$\dfrac{c \otimes d \vdash_{\mathcal{C}} e}{\langle \overline{x};\ c;\ d, \Gamma\rangle \longrightarrow \langle \overline{x};\ e;\ \Gamma\rangle}$
Ask	$\dfrac{c \vdash_{\mathcal{C}} d \otimes e}{\langle \overline{x};\ c;\ \forall \vec{z}(d \to A), \Gamma\rangle \longrightarrow \langle \overline{x};\ e;\ A[\vec{s}/\vec{z}], \Gamma\rangle}$
Persistent ask	$\dfrac{c \vdash_{\mathcal{C}} d \otimes e}{\langle \overline{x};\ c;\ \forall \vec{z}(d \Rightarrow A), \Gamma\rangle \longrightarrow \langle \overline{x};\ e;\ A[\vec{s}/\vec{z}], \forall \vec{z}(d \Rightarrow A), \Gamma\rangle}$
Hiding	$\dfrac{z \notin \overline{z} \cup \mathrm{fv}(c, \Gamma)}{\langle \overline{x};\ c;\ \exists z.A, \Gamma\rangle \longrightarrow \langle \overline{x} \cup \{z\};\ c;\ A, \Gamma\rangle}$

Table 1: Transition relation

In order to introduce the notion of predicates, Σ_C is partitioned into $\{\Sigma_D, \Sigma_{\overline{D}}\}$ such that Σ_D contains $\mathbf{1}$. Intuitively, $\Sigma_{\overline{D}}$ will contain linear tokens which should not be observed, i.e. predicates. The constraint languages formed from Σ_D and $\Sigma_{\overline{D}}$, are noted $\mathcal{D}$ and $\overline{\mathcal{D}}$ respectively.

Definition 2.7 (Observables) *Let* A *be an LCC(*$\mathcal{C}$*) agent such that* $\langle \emptyset;\ \mathbf{1};\ A\rangle \xrightarrow{*} \langle \overline{x};\ c;\ \Gamma\rangle$.

- *the constraint* $\exists \overline{x}.c$ *is an* accessible store *for* A.

- *the constraint* $\exists\overline{x}.c$ *is a* pseudo-success *for* A*, if* Γ *is a multi-set of persistent asks.*
- *the constraint* $\exists\overline{x}.d$ *is a* success *of* A*, if it is a pseudo-success for* A *such that* $\langle\overline{x};\, c;\, \Gamma\rangle \not\longrightarrow$.
- *a success* d *of* A *is a* $\mathcal{D}$-success *if* $d \in \mathcal{D}$.

Definition 2.8 (Operational Semantics)

- $\mathcal{O}^{store}(A)$ *is the set of accessible store for the agent* A.
- $\mathcal{O}^{p\text{-}s}(A)$ *is the set of pseudo-successes for the agent* A.
- $\mathcal{O}^{\mathcal{D}\text{-}succ}(A)$ *is the set of* $\mathcal{D}$*-successes for the agent* A.

2.4 Examples

The following examples illustrate, first, how usual declarations are recovered through the use of persistent ask, and then how free variables are used to provide an environment.

2.4.1 Dining Philosophers

The classical benchmark of expressiveness for concurrent languages is the *dining philosophers*. The problem consists of N philosophers sitting around a table who do nothing but think and eat. Between each of them, there is a single fork. In order to eat, a philosopher must have both the fork on his right and the one on his left. As suggested in [2], this problem has an extremely simple and elegant solution in LCC.

An even more compact solution is proposed here: the linear constraint system in this example is a combination of translation in ILL of standard equality constraint over $\mathbb{N}$ and of linear constraints token $fork/1$ and $eat/1$ with no other non-logical axioms than equality axiom schema: $c(\vec{x}) \otimes (\vec{x}{=}\vec{y}) \Vdash c(\vec{y})$ for any constraint symbol c.

Example(Dining Philosophers)
$\forall M, N.recphilo(M, N) \Rightarrow ($
$\quad fork(M)\ ||$
$\quad \forall I(fork(I) \otimes fork(I + 1 \bmod N) \Rightarrow eat(I))\ ||$
$\quad \forall I(eat(I) \Rightarrow fork(I) \otimes fork(I + 1 \bmod N))\ ||$
$\quad I \neq N \rightarrow recphilo(M + 1, N)\ )$

It is worth noting that the *philosophers* do not need to be relaunched using a recursive declaration as their encoding using persistent asks remains active.

2.4.2 Iterators

A simple iterator can be encoded thanks to the persistent asks. A more complete version is provided in Sect. 5.5 thanks to the modular constructs, which allow passing a variable associated to a persistent asks as argument of an iterator.

Example(Iterator)
$forall([]) \Rightarrow true \;||$
$forall([H|T]) \Rightarrow arg(H) \otimes forall(T) \;||$
$\forall X(arg(X) \Rightarrow Body) \;||\; forall(L)$

Here, the $forall$ persistent ask will apply the code of $Body$ (called through arg) to all the elements of the list L.

2.4.3 Dynamic Clause Assertion

In the two previous examples we have no declaration since in LCC they are replaced by persistent asks. However, this allows us to go much farther with for instance a very simple and direct encoding of dynamic clause assertions.

The straightforward recursive implementation of the Fibonacci sequence is an algorithm known to be particularly inefficient, since it computes many values repeatedly. An elegant way to improve significantly the behavior of such an algorithm is to store intermediary computed values using memoization. The computation falls from exponential to linear complexity.

As the following example shows, the use of this technique is very natural in LCC. The main idea is to use the naive recursive implementation, and to add in parallel composition with the body of the main agent, the persistent:
$\forall F'(fib(N, F') \Rightarrow F' = F))$
in which N and F are free variables, providing an environment. This agent will be in charge of consuming the (future) calls to $fib(N', F')$ asking for the computation of the N^{th} Fibonacci's number, and unify F' with the result that has already been calculated, transmitted through the variable F of the environment.

Example(Fibonacci):
$\forall N, F(fib(N, F) \otimes N < 2 \Rightarrow F = N) \;||$
$\forall N, F(fib(N, F) \otimes N > 1 \Rightarrow ($
$\quad \exists F_1, F_2.(fib(N-1, F_1) \otimes fib(N-2, F_2) \otimes F = F_1 + F_2) \;||$
$\quad \forall F'(fib(N, F') \Rightarrow F' = F))$

Despite the fact that the worst complexity of this program is still exponential, the choice of a good strategy, for example selecting first "younger" persistent ask for consuming linear tokens, leads to the result in a linear time.

From a Logic Programming (LP) point of view, the persistent ask added at the end of the clause is nothing but a dynamic clause assertion. Indeed the classical Prolog built-in `assert(p(X1,...,XN):-Body)` could be interpreted in LCC as the agent $\forall$`X1,...,XN (p(X1,...,Xn)` $\Rightarrow$ `Body)`. Moreover, variable renaming that `assert/1` made transparently, can be simply emulated by the explicit quantification provided by the LCC operator $\exists$. LCC thus provides a theoretical framework, with a first order logical semantics, to dynamic clause assertion in the context of LP. It must be noticed, however, that this implementation of `assert/1` is backtracking, i.e. that the asserted clause will be removed during the backtrack.

This idea of providing an environment through free variables (like N and F for the last persistent ask of the above example) actually encodes a closure, seen as code with an environment. Note however that using only LCC does not prevent outside code to look inside the persistent ask, which leads us to provide *code protection* through a system of modules, seen as restrictions on the possible scope of some variables. Moreover, *modules* will provide simple tools to attach a variable to a persistent ask, and thus permit to pass a persistent ask as the argument of another call.

3 Modular LCC

3.1 Modular Constraint Systems

Let $\mathcal{C}$ be a constraint system. To introduce the notion of modules, we suppose that Σ_C is further partitioned into $\{\Sigma_G, \Sigma_M\}$ such that Σ_G contains $=$ and $\mathbf{1}$. The constraints formed from Σ_G (resp. Σ_M) form the language $\mathcal{G}$ (resp. $\mathcal{M}$) of *built-in constraints* (resp. *modular constraints*). Possibly banged atomic constraints in $\mathcal{G}$ and $\mathcal{M}$ are noted g and m respectively. c will be a notation for any constraint in $\mathcal{C}$.

3.2 Syntax of MLCC($\mathcal{C}$)

The syntax of MLCC extends the one of LCCwith a localization operator of an agent is a module:

Definition 3.1 *The syntax of MLCC($\mathcal{C}$)* agents *is given by the following grammar:*
$A ::= t\{A\} \mid t{:}c \mid A||A \mid \exists x.A \mid \forall\vec{x}(c \rightarrow A) \mid \forall\vec{x}(c \Rightarrow A)$

The new constructs $t\{A\}$ stands for the localization of agent A in the module t. The *tell* agent has now a new form: $t{:}c$, corresponding intuitively to adding the constraint c of $\mathcal{C}$ in the module *named* by the term t of $\mathcal{T}$.

3.3 Modular Constraints

The modularization of MLCC agents introduces a modularization of the constraint store.

Definition 3.2 (Modular Store) *A* prefixed constraint $\mathtt{m} = t{:}m$ *is an atomic constraint (possibly banged)* m *of* $\mathcal{M}$ *prefixed by a term* t *of* $\mathcal{T}$*,* $t{:}\overline{m}$ *will be a notation for* $t{:}m_1, \ldots, t{:}m_k$ *if* $\overline{m} = m_1, \ldots, m_k$*.*

A modular store *is a formula* $\exists\overline{x}.(g|\overline{\mathtt{m}})$ *where* g *is a conjunction of constraints of* $\mathcal{G}$ *without quantification and* $\overline{\mathtt{m}}$ *a multi-set of prefixed constraints.*

In the following we will use, $\mathbb{c}$ or $\mathbb{d}$ to note modular stores and $\mathbb{1}$ to note the modular empty store $(\mathbf{1}|\emptyset)$.

Definition 3.3 *We define an order on modular stores as follows:*

$$\textbf{transitivity} \qquad \frac{\mathbb{c} >_{\mathcal{C}} \mathbb{d} \quad \mathbb{d} >_{\mathcal{C}} \mathbb{c}'}{\mathbb{c} >_{\mathcal{C}} \mathbb{c}'}$$

$$\textbf{substitution} \qquad \frac{g \vdash_{\mathcal{C}} t{=}t' \otimes g'}{\exists\overline{x}.(g|\overline{\mathtt{m}}, t{:}m) >_{\mathcal{C}} \exists\overline{x}.(g'|\overline{\mathtt{m}}, t'{:}m)}$$

$$\textbf{entailment} \qquad \frac{g \otimes \bigotimes \overline{m} \vdash_{\mathcal{C}} g' \otimes \bigotimes \overline{m}'}{\exists\overline{x}.(g|\overline{\mathtt{m}}, t{:}\overline{m}) >_{\mathcal{C}} \exists\overline{x}.(g'|\overline{\mathtt{m}}, t{:}\overline{m}')}$$

By abuse of notation, we extend the tensor product of linear constraints to modular stores:

Definition 3.4 *The* conjunction *of two modular stores* $\mathbb{c} = \exists\overline{x}.(g|\overline{\mathtt{m}})$ *and* $\mathbb{c}' = \exists\overline{x}'.(g'|\overline{\mathtt{m}}')$ *is the store* $(\mathbb{c} \otimes \mathbb{c}') = \exists\overline{x}, \overline{x}'.(g \otimes g'|\overline{\mathtt{m}}, \overline{\mathtt{m}}')$ *if* $\overline{x} \cap \overline{x}' = \emptyset$*.*

Lemma 3.5 (Monotonicity of $\otimes$) *For all modular stores* $\mathbb{c}$*,* $\mathbb{d}$ *and* $\mathbb{d}'$ *if* $\mathbb{d} >_{\mathcal{C}} \mathbb{d}'$ *then* $\mathbb{c} \otimes \mathbb{d} >_{\mathcal{C}} \mathbb{c} \otimes \mathbb{d}'$

Proof: By induction on the proof π of $(g'|\overline{\mathtt{m}}') >_{\mathcal{C}} (g''|\overline{\mathtt{m}}'')$ we prove that $(g \otimes g'|\overline{\mathtt{m}}, \overline{\mathtt{m}}') >_{\mathcal{C}} (g \otimes g''|\overline{\mathtt{m}}, \overline{\mathtt{m}}'')$. In this proof we suppose that all $\overline{\mathtt{m}}$'s are not empty, if it is not the case, just recall that $c \otimes \mathbf{1} \vdash c \vdash c \otimes \mathbf{1}$.

- π ends with transitivity: trivial.
- π ends with *substitution*:

  $$\frac{g' \vdash_{\mathcal{C}} t{=}t' \otimes g''}{\exists\overline{x}.(g'|\overline{\mathtt{m}}, t{:}m) >_{\mathcal{C}} \exists\overline{x}.(g''|\overline{\mathtt{m}}, t'{:}m)}$$

  Thank to $\otimes$*-left* rule, we infer that $g \otimes g' \vdash_{\mathcal{C}} g \otimes t = t' \otimes g''$ and then conclude immediately.

- π ends with *entailment*:

  $$\frac{g' \otimes \bigotimes \overline{m}' \vdash_{\mathcal{C}} g'' \otimes \bigotimes \overline{m}''}{\exists\overline{x}.(g'|\overline{\mathtt{m}}, t{:}\overline{m}') >_{\mathcal{C}} \exists\overline{x}.(g''|\overline{\mathtt{m}}, t{:}\overline{m}'')}$$

Thank to $\otimes$-*left* rule, we infer that

$$g \otimes g' \otimes \bigotimes \overline{m}' \vdash_{\mathcal{C}} g \otimes g'' \otimes \bigotimes \overline{m}''$$

, and hence conclude.

□

3.4 Operational Semantics

We will now provide a precise operational semantics to MLCC, based as usual on a notion of configuration, through a transition relation and a structural congruence.

Definition 3.6 (Configuration) *A* configuration *is a tuple* $\langle \overline{x};\ \exists \overline{y}.(g|\overline{\mathrm{m}});\ \Gamma \rangle$ *where* $\overline{x}$ *is a multi-set of variable,* Γ *a multi-set of localized agents and* $\exists \overline{y}.(g|\overline{\mathrm{m}})$ *a modular store such that* $\overline{y} \cap \mathrm{fv}(\Gamma, \overline{x}) = \emptyset$

Definition 3.7 *The structural congruence* $\equiv$ *is the least congruence satisfying the following rule of* parallel composition*:*

$$\langle \overline{x};\ \mathbb{c};\ t\{A \,||\, B\}, \Gamma \rangle \equiv \langle \overline{x};\ \mathbb{c};\ t\{A\}, t\{B\}, \Gamma \rangle$$

Definition 3.8 *The transition relation* $\longrightarrow$ *is the least relation satisfying the rules presented in the table 2.*

This operational semantics enjoys the same kind of properties as the original LCC operational semantics.

Proposition 3.9 (Monotonicity) *For every derivation* $\delta = ((\overline{x}; \mathbb{c}; \Gamma) \xrightarrow{*} (\overline{x}'; \mathbb{c}'; \Gamma'))$*, there exists* $\overline{y}$ *free in* δ*,* Δ *, and a modular store* $\mathbb{d}$ *such as* $(\overline{x}, \overline{y}; \mathbb{c} \otimes \mathbb{d}; \Gamma, \Delta) \xrightarrow{*} (\overline{x}', \overline{y}; \mathbb{c}' \otimes \mathbb{d}; \Gamma', \Delta)$*.*

Proof: By induction on the derivation δ:

- For *equivalence* it is trivial.
- For *tell* just note that thanks to the monotonicity of $\otimes$, if $c \otimes \exists \overline{y}.(g|\overline{\mathrm{m}}) >_{\mathcal{C}} \mathbb{c}'$ then $c \otimes \exists \overline{y}.(g|\overline{\mathrm{m}}) \otimes \mathbb{d} >_{\mathcal{C}} \mathbb{c}' \otimes \mathbb{d}$.
- For *ask* and *persistent ask* note that thanks to the monotonicity of $\otimes$, if $\mathbb{c} >_{\mathcal{C}} \exists \overline{y}.(g|\overline{\mathrm{m}}, t : \overline{m})$ and $d >_{\mathcal{C}} \exists \overline{y}'.(g'|\overline{\mathrm{m}}')$ then $\mathbb{c} \otimes \mathbb{d} >_{\mathcal{C}} \exists \overline{y}, \overline{y}'.(g \otimes g'|\overline{\mathrm{m}}, \overline{\mathrm{m}}', t : \overline{m})$ and that if $\bigotimes g \otimes \bigotimes \overline{m} \vdash \bigotimes g'' \otimes \bigotimes \overline{m}'' \otimes d[\vec{s}/\vec{z}]$ then $\bigotimes(g \otimes g') \otimes \bigotimes \overline{m} \vdash \bigotimes(g'' \otimes g') \otimes \bigotimes \overline{m}'' \otimes d[\vec{s}/\vec{z}]$
- For *hiding* one just use the α-conversion to be sure that $\overline{y}$ is free in δ.
- For other rules notice that they can be done in $(\overline{x}\,\overline{y}; \mathbb{c} \otimes \mathbb{d}; \Gamma, \Delta)$ since they do not have condition about the hidden variables or the store.

Equivalence	$\dfrac{\langle \overline{x};\, \mathbb{c};\, \Gamma\rangle \equiv \langle \overline{x};\, \mathbb{c}';\, \Gamma'\rangle \longrightarrow \langle \overline{y};\, \mathbb{d}';\, \Delta'\rangle \equiv \langle \overline{y};\, \mathbb{d};\, \Delta\rangle}{\langle \overline{x};\, \mathbb{c};\, \Gamma\rangle \longrightarrow \langle \overline{y};\, \mathbb{d};\, \Delta\rangle}$
Modularize	$\langle \overline{x};\, \mathbb{c};\, t\{s\{A\}\}, \Gamma\rangle \longrightarrow \langle \overline{x};\, \mathbb{c};\, s\{A\}, \Gamma\rangle$
Tell	$\dfrac{d \vdash \exists \overline{y}.(g \otimes \bigotimes \overline{m}) \quad \mathbb{c} \otimes \exists \overline{y}.(g \mid t{:}\overline{m}) >_{\mathcal{C}} \mathbb{c}'}{\langle \overline{x};\, \mathbb{c};\, s\{t{:}d\}, \Gamma\rangle \longrightarrow \langle \overline{x};\, \mathbb{c}';\, \Gamma\rangle}$
Ask	$\dfrac{\mathbb{c} > \exists \overline{y}.(g \mid \overline{\mathbb{m}}, t{:}\overline{m}) \qquad g \otimes \bigotimes \overline{m} \vdash g' \otimes \bigotimes \overline{m}' \otimes d[\vec{s}/\vec{z}]}{\langle \overline{x};\, \mathbb{c};\, t\{\forall \vec{z}(d \to A)\}, \Gamma\rangle \longrightarrow \langle \overline{x};\, \exists \overline{y}.(g' \mid \overline{\mathbb{m}}, t{:}\overline{m}');\, t\{A[\vec{s}/\vec{z}]\}, \Gamma\rangle}$
Persistent Ask	$\dfrac{\mathbb{c} > \exists \overline{y}.(g \mid \overline{\mathbb{m}}, t{:}\overline{m}) \qquad g \otimes \bigotimes \overline{m} \vdash g' \otimes \bigotimes \overline{m}' \otimes d[\vec{s}/\vec{z}]}{\langle \overline{x};\, \mathbb{c};\, t\{\forall \vec{z}(d \Rightarrow A)\}, \Gamma\rangle \longrightarrow \langle \overline{x};\, \exists \overline{y}.(g' \mid \overline{\mathbb{m}}, t{:}\overline{m}');\, t\{A[\vec{s}/\vec{z}]\}, t\{\forall \vec{z}(d \Rightarrow A)\}, \Gamma\rangle}$
Hiding	$\dfrac{z \notin \overline{x} \cup \mathrm{fv}(\mathbb{c}, \Gamma, t)}{\langle \overline{x};\, \mathbb{c};\, t\{\exists z.A\}, \Gamma\rangle \longrightarrow \langle \overline{x} \cup \{z\};\, \mathbb{c};\, t\{A\}, \Gamma\rangle}$

Table 2: Transition relation

□

The observables of interest for MLCC are defined as previously by replacing "constraints" by "modular stores", where generally, $\mathcal{D}$ is chosen equal to $\mathcal{G}$:

Definition 3.10 (Observables) *Let A be an MLCC(C) agent such that $\langle \emptyset;\, \mathbb{1};\, x\{A\}\rangle \xrightarrow{*} \langle \overline{y};\, \mathbb{c};\, \Gamma\rangle$ for some $x \notin \mathrm{fv}(A)$.*

- *the modular store $\exists \overline{y}.\mathbb{c}$ is an* accessible store *for A.*
- *the modular store $\exists \overline{y}.\mathbb{c}$ is a* pseudo-success *for A, if Γ is a multi-set of persistent asks.*
- *the modular store $\exists \overline{y}.\mathbb{c}$ is a* success *of A, if it is a pseudo-success for A such that $\langle \overline{y};\, \mathbb{c};\, \Gamma\rangle \not\longrightarrow$.*
- *The modular store $\exists \overline{y}.(g \mid \overline{\mathbb{m}})$ is a* $\mathcal{D}$-success *for A, if it is a success for A such that $\overline{\mathbb{m}} = \emptyset$ and $\exists y.g \in \mathcal{D}$*

3.5 Example: Beyond Dining Philosophers

Let us improve on the example of Sect. 2.4.1 in order to demonstrate the expressive power gained from the *modular* constructs and from the *persistent ask*.

The module constructs allow to extend the dining philosophers' example to a "banquet" of several tables of philosophers, where each table is an independent module. The corresponding MLCC agent below creates N tables of P philosophers:

Example(Banqueting Philosophers).

$$
\begin{array}{l}
banquet\{ \\
\quad \forall I, N, P.recTable(I, N, P) \Rightarrow \\
\qquad \exists Table.Table\{ \\
\qquad\quad \forall J.recPhilo(J) \Rightarrow (\\
\qquad\qquad Table : fork(J) \;||\; \\
\qquad\qquad fork(J) \otimes fork(J+1 \;mod\; P) \Rightarrow \\
\qquad\qquad\quad Table : eat(J) \;||\; \\
\qquad\qquad eat(J) \Rightarrow \\
\qquad\qquad\quad Table : (fork(J) \otimes fork(J+1 \;mod\; P)) \;||\; \\
\qquad\qquad J \neq P \rightarrow Table : recPhilo(J+1)\;) \;||\; \\
\qquad\quad Table : recPhilo(0) \\
\qquad \} \;|| \\
\qquad I \neq N \rightarrow banquet : recTable(I+1, N, P) \\
\}
\end{array}
$$

Since the logical semantics of MLCC enjoys the same correction properties than that of LCC (see theorem 4.4 below), the phase semantics of Linear Logic can be used to prove safety properties in way similar to [6], such as for instance here, that no philosopher can use a fork belonging to another table.

3.6 Code Protection

One important feature of a module system is its capability to hide implementations and guarantee the protection of module code. In MLCC, the *code protection* property means that if a module $t\{\exists x(x\{A\} \;||\; B)\}$ is composed of an interface B and a hidden implementation A, then a parallel agent C cannot add any constraint of the form $x : c$ nor unblock any of its ask with such a constraint. This leads to the following property:

Proposition 3.11 (Code protection) *Let A, B and C be three MLCC agents, and t a term of T. Let*

$$M = t\{\exists x(x\{A\} \;||\; B)\} \;||\; C$$

If A and B do not add any constraint on x to the store $\mathbb{c}$, except those of the form $x : c$, then C cannot add any constraint of the form $x : c$ nor unblock any of its ask with such a constraint in a derivation from M.

Proof: We will suppose that x is not free in C nor in $\mathbb{c}$. If that is not the case, then x (the one under the $\exists$) will be renamed by α-conversion in order to use the **Hiding** rule.

We thus have a configuration of the form: $(\overline{x} \cup \{x\}; \mathbb{c}; x\{A\}, t\{B\}, C)$, such that $x \notin \mathrm{fv}(C, \mathbb{c}, t)$. Let us prove that as long as A and B do not add constraints on x except those of the form $x:c$, x will remain bound in C and thus C will not be able to tell nor ask any constraint on x. This is indeed enough since the restriction on A and B forbids that any ask (resp. tell) on another term is unblocked by (resp. unblocks) a tell (resp. an ask) on x since $x:c$ will never imply a constraint like $x = t$ with x bound in t.

We only need to prove this property for one step of derivation, it will then hold for any finite derivation by induction. Let us consider all the cases of derivation. If $x\{A\}$ or $t\{B\}$ are the chosen agents, then the property trivially holds since C did not change. If C is the agent chosen for derivation, the rules **Modularize** and **Tell** obviously don't change anything w.r.t. x being bound in C. The **Hiding** rule might make a bound variable free, but since we have $\{x\} \cup \overline{x}$ as first member of our configuration, we know that the **Hiding** rule will only apply to another variable. The case of the **Equivalence** rule is treated by induction on the equivalent configurations. For the **Ask** and **Persistent Ask** rules, the only risk is that the renaming of the variables under $\forall$ replaces some of them by a term containing x. However remember that the only replacement happens on variables appearing in a linear token d, when $\mathbb{c} \vdash \mathbb{c}' \otimes d[t(x)/y]$. From the lemma below and knowing that x is bound in $\mathbb{c}$, the above implication with x free in t is impossible, i.e. x remains bound after an **Ask** or a **Persistent Ask** rule. □

Lemma 3.12 *If $x \in \mathrm{fv}(m)$ such as m is linear token and $c \vdash_{\mathcal{C}} m \otimes d$ then $x \in \mathrm{fv}(c)$*

Proof: By induction on the proof π of $c \vdash_{\mathcal{C}} m \otimes d$ where d is an arbitrary constraint. Just recall that we have supposed in the definition of $\mathcal{C}$ that all free variables occurring in the right hand side of a non-logical axiom appears in its left hand side. □

4 Logical Semantics

One striking feature of LCC languages is their simple semantics in Linear Logic [6, 16, 18] allowing for various proof methods coming from Linear Logic. In this section, we generalize the results of [6] to the richer fragment of LL containing banged implications as used in MLCC programs.

Definition 4.1 *In a module t, constraints, agents and store are translated into formulas in the following way (in the following we suppose with no loss of generality that $x \notin \mathrm{fv}(t)$ and $\overline{x} \cap \mathrm{fv}(t) = \emptyset$):*

$$(c \otimes d)^t = c^t \otimes d^t$$

$$(\exists x.c)^t = \exists x.c^t \qquad g(s_1, \ldots, s_n)^t = g(s_1, \ldots, s_n)$$

$$(!c)^t = !c^t \qquad m(s_1, \ldots, s_n)^t = \dot{m}(t, s_1, \ldots, s_n)$$

$$(\exists x.A)^t = \exists x.A^t$$
$$s\{A\}^t = A^s \qquad (s:c)^t = c^s$$
$$(A \mid\mid B)^t = A^t \otimes B^t \qquad (\forall \overline{x}(c \to A))^t = \forall \overline{x}(c^t \multimap A^t)$$
$$(\forall \overline{x}(c \Rightarrow A))^t = !\forall \overline{x}(c^t \multimap A^t)$$

For any multi-set $\Gamma = (\gamma_1, \ldots, \gamma_n)$ *of agents or prefixed constraints we define* $\Gamma^t = {\gamma_1}^t \otimes \cdots \otimes {\gamma_n}^t$ *and* $\emptyset^t = \mathbf{1}$. *Finally agents, Stores and Configurations are translated into formulae in the following way, where* $x \in \mathrm{fv}(A, \mathbb{c}, \Gamma)$*:*

$$\exists \overline{y}.\, (g | \biguplus_i \{t_i : m_i\})^\dagger = \exists \overline{y}.\, \left(g \otimes \bigotimes\nolimits_i {m_i}^{t_i}\right)$$
$$A^\dagger = A^x \qquad \langle \overline{y};\, \mathbb{c};\, \Gamma \rangle^\dagger = \exists \overline{y}.(\mathbb{c}^\dagger \otimes \Gamma^x)$$

$(\mathcal{C}^\dagger, \Vdash_{\mathcal{C}^\dagger})$ *is the constraint system formed from* $(\Sigma_G \uplus \dot{\Sigma}_M)$, Σ_T *and* V *such that iff* $c_1, \ldots, c_n \Vdash_{\mathcal{C}} c$ *then* $c_1^x, \ldots, c_n^x \Vdash_{\mathcal{C}^\dagger} c^x$ *with* $x \notin \mathrm{fv}(c, c_1, \ldots, c_n)$ *and that for all* $\dot{m} \in \dot{\Sigma}_M$ $\dot{m}(x, \vec{z}), !x = y \vdash_{\mathcal{C}^\dagger} \dot{m}(y, \vec{z})$.

Lemma 4.2 *Let* Γ *be a sequence of constraints,* c *be a constraint and* x *be a variable free in* Γ *and* c*, if* $\Gamma \vdash_{\mathcal{C}} c$ *then* $\Gamma^x \vdash_{\mathcal{C}^\dagger} c^x$.

Proof: By induction on the proof of $\Gamma \vdash_{\mathcal{C}} c$. □

Lemma 4.3 (Soundness of $>_{\mathcal{C}}$) *For all modular stores* $\mathbb{c}$ *and* $\mathbb{d}$ *if* $\mathbb{c} >_{\mathcal{C}} \mathbb{d}$ *then* $\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{d}^\dagger$.

Proof: As previously we suppose that all $\overline{g}$'s and all $\overline{\mathrm{m}}$'s are not empty, if it is not the case that just recall that $c \otimes \mathbf{1} \vdash c \vdash c \otimes \mathbf{1}$.

By induction on the proof π of $\mathbb{c} >_{\mathcal{C}} \mathbb{d}$:

- π ends with *transitivity* rules:

$$\frac{\mathbb{c} >_{\mathcal{C}} \mathbb{c}' \quad \mathbb{c}' >_{\mathcal{C}} \mathbb{d}}{\mathbb{c} >_{\mathcal{C}} \mathbb{d}}$$

  By induction hypothesis, $\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{c}'^\dagger$ and $\mathbb{c}'^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{d}^\dagger$, then thanks to *cut* rule, we have $\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{d}^\dagger$.

- π ends with *substitution*:

$$\frac{g \vdash_{\mathcal{C}} t = t' \otimes g'}{\exists \overline{x}.(g|\overline{\mathrm{m}}, t:m) >_{\mathcal{C}} \exists \overline{x}.(g'|\overline{\mathrm{m}}, t':m)}$$

$$\frac{\dfrac{\dfrac{g \vdash_{\mathcal{C}} t = t' \otimes g'}{g \vdash_{\mathcal{C}^\dagger} t = t' \otimes g'}\ \mathrm{l}\ 4.2}{g \otimes m^t \vdash_{\mathcal{C}^\dagger} t = t' \otimes g' \otimes m^t}\ \otimes\text{-r} \quad \dfrac{x = y' \otimes m^x \vdash_{\mathcal{C}^\dagger} m^y}{t = t' \otimes m^t \vdash_{\mathcal{C}^\dagger} m^{t'}}\ \forall}{g \otimes m^t \vdash_{\mathcal{C}^\dagger} g' \otimes m^{t'}}\ \mathrm{cut}$$

- π ends with *entailment*:

$$\frac{g \otimes \bigotimes \overline{m} \vdash_{\mathcal{C}} g' \otimes \bigotimes \overline{m}'}{\exists \overline{x}.(g|\overline{\mathrm{m}}, t{:}\overline{m}) >_{\mathcal{C}} \exists \overline{x}.(g'|\overline{\mathrm{m}}, t{:}\overline{m}')}$$

$$\dfrac{\dfrac{\dfrac{g \otimes \bigotimes \overline{m} \vdash_{\mathcal{C}} g' \otimes \bigotimes \overline{m}'}{g \otimes \bigotimes \overline{m}^t \vdash_{\mathcal{C}\dagger} g' \otimes \bigotimes \overline{m}'^t}\ \text{l 4.2}}{g \otimes \bigotimes \overline{\mathrm{m}}^\dagger \otimes \bigotimes \overline{m}^t \vdash_{\mathcal{C}\dagger} g' \otimes \bigotimes \overline{\mathrm{m}}^\dagger \otimes \bigotimes \overline{m}'^t}\ \otimes\text{-R}}{\exists \overline{x}.(g \otimes \bigotimes \overline{\mathrm{m}}^\dagger \otimes \bigotimes \overline{m}^t) \vdash_{\mathcal{C}\dagger} \exists \overline{x}.(g' \otimes \bigotimes \overline{\mathrm{m}}^\dagger \otimes \bigotimes \overline{m}'^t)}\ \exists$$

□

Theorem 4.4 (Soundness) *Let κ and κ' be two configurations.*
If $\kappa \equiv \kappa'$ then $\kappa^\dagger \dashv\vdash_{\mathcal{C}\dagger} \kappa'^\dagger$
If $\kappa \xrightarrow{*} \kappa'$ then $\kappa^\dagger \vdash_{\mathcal{C}\dagger} \kappa'^\dagger$

Proof: By induction on $\equiv$ and $\xrightarrow{*}$:

- for *parallel composition*, *equivalence* and *modularize* it is immediate;
- for *hiding*, $\exists x.(A \otimes B) \dashv\vdash A \otimes \exists x.B$ and $\exists x.A \dashv\vdash A$ if $x \notin \mathrm{fv}(A)$;
- for *tell*:

$$\frac{d \vdash g \otimes \bigotimes \overline{\mathrm{m}} \quad \mathbb{c} \otimes (g|\overline{\mathrm{m}}) >_{\mathcal{C}} \mathbb{c}'}{\langle \overline{x};\ \mathbb{c};\ s\{t{:}d\}, \Gamma\rangle \longrightarrow \langle \overline{x};\ \mathbb{c}';\ \Gamma\rangle}$$

$$\dfrac{\dfrac{\dfrac{d \vdash_{\mathcal{C}} g \otimes \bigotimes \overline{m}'}{d^t \vdash_{\mathcal{C}\dagger} g \otimes \bigotimes \overline{m}'^t}\ \text{l 4.2} \quad \dfrac{\mathbb{c} \otimes (g|\overline{\mathrm{m}}) >_{\mathcal{C}} \mathbb{c}'}{\mathbb{c}^\dagger \otimes g \otimes \bigotimes \overline{m}'^t \vdash_{\mathcal{C}\dagger} \mathbb{c}'^\dagger}\ \text{l 4.3}}{\mathbb{c}^\dagger \otimes d^t \vdash_{\mathcal{C}\dagger} \mathbb{c}'^\dagger}\ \textit{cut}}{\exists \overline{x}.(\mathbb{c}^\dagger \otimes d^t \otimes \Gamma^\dagger) \vdash_{\mathcal{C}\dagger} \exists \overline{x}.(\mathbb{c}'^\dagger \otimes \Gamma^\dagger)}\ \exists, \otimes$$

- for *ask*:

$$\frac{\mathbb{c} >_{\mathcal{C}} \exists \overline{y}.(g|\overline{\mathrm{m}}, t{:}\overline{m}) \quad g \otimes \bigotimes \overline{m} \vdash_{\mathcal{C}} g' \otimes \bigotimes \overline{m}' \otimes d[\vec{s}/\vec{z}]}{\begin{array}{c}\langle \overline{x};\ \mathbb{c};\ t\{\forall \vec{z}(d \to A)\}, \Gamma\rangle \longrightarrow \\ \langle \overline{x};\ \exists \overline{y}.(g'|\overline{\mathrm{m}}, t{:}\overline{m}');\ t\{A[\vec{s}/\vec{z}]\}, \Gamma\rangle\end{array}}$$

  First of all notice that if $\overline{y} \cap \mathrm{fv}(t) = \emptyset$ then $(A[\overline{s}/\overline{y}])^t = A^t[\overline{s}/\overline{y}]$. Now let $\mathbb{c}' = (g|\overline{\mathrm{m}}, t{:}\overline{m})$, $\mathbb{c}'' = (g'|\overline{\mathrm{m}}, t{:}\overline{m}')$ and $B = (d^t \multimap A^t)$.

$$\pi_1 = \dfrac{\dfrac{g \otimes \bigotimes \overline{m} \vdash_{\mathcal{C}} g' \otimes \bigotimes \overline{m}' \otimes d[\vec{s}/\vec{z}]}{g \otimes \bigotimes \overline{m}^t \vdash_{\mathcal{C}\dagger} g' \otimes \bigotimes \overline{m}'^t \otimes d^t[\vec{s}/\vec{z}]}\ \text{l 4.2}}{\mathbb{c}'^\dagger \vdash_{\mathcal{C}\dagger} \mathbb{c}''^\dagger \otimes d^t[\vec{s}/\vec{z}]}\ \otimes$$

$$\pi_2 = \dfrac{\mathbb{c} >_{\mathcal{C}} \exists \overline{y}.\mathbb{c}'}{\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} \exists \overline{y}.\mathbb{c}'^\dagger}\ \mathrm{l}\ 4.3$$

$$\dfrac{\pi_2 \quad \dfrac{\pi_1 \quad \dfrac{}{\mathbb{c}''^\dagger \otimes d^t[\vec{s}/\vec{z}], B[\vec{s}/\vec{z}] \vdash_{\mathcal{C}^\dagger} \mathbb{c}''^\dagger \otimes A^t[\vec{s}/\vec{z}]}\ \otimes,\multimap}{\mathbb{c}'^\dagger, B[\vec{s}/\vec{z}], \vdash_{\mathcal{C}^\dagger} \mathbb{c}''^\dagger \otimes A^t[\vec{s}/\vec{z}]}\ \text{cut}}{\mathbb{c}^\dagger, \forall \overline{z}.B, \vdash_{\mathcal{C}^\dagger} \exists \overline{y}.\mathbb{c}''^\dagger \otimes A^t[\vec{s}/\vec{z}]}\ \exists,\forall}{\exists \overline{y}.\mathbb{c}'^\dagger, \forall \overline{z}.B, \vdash_{\mathcal{C}^\dagger} \exists \overline{y}.\mathbb{c}''^\dagger \otimes A^t[\vec{s}/\vec{z}]}\ \text{cut}}{\exists \overline{x}.(\mathbb{c}^\dagger \otimes \forall \overline{z}.B \otimes \Gamma^\dagger) \vdash_{\mathcal{C}^\dagger} \exists \overline{x}.(\exists \overline{y}.\mathbb{c}''^\dagger \otimes A^t[\vec{s}/\vec{z}] \otimes \Gamma^\dagger)}\ \exists,\otimes$$

- for *persistent ask*:

$$\dfrac{\mathbb{c} > \exists \overline{y}.(g|\overline{\mathrm{m}}, t{:}\overline{m}) \qquad g \otimes \bigotimes \overline{m} \vdash g' \otimes \bigotimes \overline{m}' \otimes d[\vec{s}/\vec{z}]}{\begin{array}{c}\langle \overline{x};\ \mathbb{c};\ t\{\forall \vec{z}(d \Rightarrow A)\}, \Gamma\rangle \longrightarrow \\ \langle \overline{x};\ \exists \overline{y}.(g'|\overline{\mathrm{m}}, t{:}\overline{m}');\ t\{A[\vec{s}/\vec{z}]\}, t\{\forall \vec{z}(d \Rightarrow A)\}, \Gamma\rangle\end{array}}$$

  By using the previous notation for $\mathbb{c}', \mathbb{c}''$ and B we have:

$$\dfrac{\dfrac{\dfrac{}{\mathbb{c}^\dagger, \forall \overline{z}.B, !\forall \overline{z}.B, \Gamma^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{c}^\dagger \otimes \forall \overline{z}.B \otimes !\forall \overline{z}.B \otimes \Gamma^\dagger}\ \otimes}{\mathbb{c}^\dagger, !\forall \overline{z}.B, \Gamma^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{c}^\dagger \otimes \forall \overline{z}.B \otimes !\forall \overline{z}.B \otimes \Gamma^\dagger}\ !}{\exists \overline{x}.(\mathbb{c}^\dagger \otimes !\forall \overline{z}.B \otimes \Gamma^\dagger) \vdash_{\mathcal{C}^\dagger} \exists \overline{x}.(\mathbb{c}^\dagger \otimes \forall \overline{z}.B \otimes !\forall \overline{z}.B \otimes \Gamma^\dagger)}\ \exists,\otimes$$

  By using the result of the previous case we can conclude.

□

Lemma 4.5 (Completeness of $>_{\mathcal{C}}$) *For all modular stores* $\mathbb{c}$ *and* $\mathbb{d}$*, if* $\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} \mathbb{d}^\dagger$ *then* $\mathbb{c} >_{\mathcal{C}} \mathbb{d}$.

Proof: Let $()^{-\dagger}$ be a partial translation of multi-sets of constraints into non quantified stores defined as follow:

$$\begin{array}{rclcrcl} g^{-\dagger} & = & (g|\emptyset) & & \dot{m}(t,\vec{s})^{-\dagger} & = & (\emptyset|t{:}m(\vec{s})) \\ (!g)^{-\dagger} & = & (!g|\emptyset) & & (!\dot{m}(t,\vec{s}))^{-\dagger} & = & (\emptyset|!t{:}m(\vec{s})) \\ (c \otimes c')^{-\dagger} & = & c^{-\dagger} \otimes c'^{-\dagger} & & (\Gamma, \Delta)^{-\dagger} & = & \Gamma^{-\dagger} \otimes \Delta^{-\dagger} \end{array}$$

First we prove the intermediary result: for every multi-set of conjunctions of constraints Γ and every conjunction of constraints d if $\Gamma \vdash_{\mathcal{C}^\dagger} d$ then $\Gamma^{-\dagger} >_{\mathcal{C}} d^{-\dagger}$. We prove this result by induction on the proof π of the sequent $\Gamma \vdash_{\mathcal{C}^\dagger} d$.

- π is an *axiom* of the form $\Gamma^x \vdash_{\mathcal{C}^\dagger} d^x$ such as $\Gamma \vdash_{\mathcal{C}} d$:
  let $(\Gamma^x)^{-\dagger} = (g|x{:}\overline{m})$ and $(d^x)^{-\dagger} = (g'|x{:}\overline{m}')$. Now just use the *entailment* rule to conclude.

- π is an *axiom* of the form $\dot{m}(x,\vec{z}), x = y \vdash_{\mathcal{C}^\dagger} \dot{m}(y,\vec{z})$: in such a case just use the *substitution* rule.
- π ends with *cut*:
$$\frac{\Gamma \vdash_{\mathcal{C}^\dagger} c \qquad c, \Delta \vdash_{\mathcal{C}^\dagger} d}{\Gamma, \Delta \vdash_{\mathcal{C}^\dagger} d}$$
By induction hypothesis, $\Gamma^{-\dagger} >_{\mathcal{C}} c^{-\dagger}$ and $c^{-\dagger} \otimes \Delta^{-\dagger} >_{\mathcal{C}} d^{-\dagger}$. Thanks to monotonicity of $\otimes$ (lemma 3.5) and using the *transitivity* rule we can conclude that $\Gamma^{-\dagger} \otimes \Delta^{-\dagger} >_{\mathcal{C}} d^{-\dagger}$.
- π ends with $\otimes$-left: trivial
- π ends with $\otimes$-right:
$$\frac{\Gamma \vdash_{\mathcal{C}^\dagger} c \quad \Delta \vdash_{\mathcal{C}^\dagger} d}{\Gamma, \Delta \vdash_{\mathcal{C}^\dagger} c \otimes d}$$
By induction hypothesis $\Gamma^{-\dagger} >_{\mathcal{C}} c^{-\dagger}$ and $\Delta^{-\dagger} >_{\mathcal{C}} d^{-\dagger}$. By using the monotonicity of $\otimes$ (lemma 3.5) we have $\Gamma^{-\dagger} \otimes \Delta^{-\dagger} >_{\mathcal{C}} c^{-\dagger} \otimes \Delta^{-\dagger}$ and $c^{-\dagger} \otimes \Delta^{-\dagger} >_{\mathcal{C}} c^{-\dagger} \otimes d^{-\dagger}$. By using the *transitivity* rule we have finally $\Gamma^{-\dagger} \otimes \Delta^{-\dagger} >_{\mathcal{C}} c^{-\dagger} \otimes d^{-\dagger}$.
- π ends with one of the four rules for !: Just notice that the four following sequents are true:
  - $c \otimes d \vdash c \otimes !d$ for *dereliction*;
  - $!c \vdash c$ for *promotion*;
  - $c \vdash c \otimes !d$ for *weakening*;
  - $c \otimes !d \otimes !d \vdash c \otimes !d$ for *contraction*.

Now it is easy to prove by an induction on $\mathbb{c}$ that for every non quantified store $\mathbb{c}$, we have $(\mathbb{c}^\dagger)^{-\dagger}$. Hence we prove the result on non quantified store. From here we can conclude easily by noting that if $c \vdash d$ then $\exists x.c \vdash \exists x.d$ □

Lemma 4.6 *For any constraint c of $\mathcal{C}$, there exists a set of variables $\overline{x}$ not free in c and constraint without quantification g of $\mathcal{G}$ and a multi-set of atomic constraints (possibly banged) $\overline{m}$ of $\mathcal{M}$ such that $c \dashv\vdash \exists \overline{x}.g \otimes \bigotimes \overline{m}$*

Proof: By induction on c:

- c is an atomic constraint (possibly banged) of $\mathcal{G}$: trivial.
- c is an atomic constraint (possibly banged) of $\mathcal{M}$: $c \dashv\vdash \mathbf{1} \otimes c$.
- $c = c' \otimes c''$: By induction hypothesis we have $c' \dashv\vdash \exists \overline{x}'.(g' \otimes \bigotimes \overline{m}')$ and $c'' \dashv\vdash \exists \overline{x}''.(g'' \bigotimes \overline{m}'')$. We can suppose without lost of generality that $\overline{x}' \cap \mathrm{fv}(c'') = \emptyset$ and $\overline{x}'' \cap \mathrm{fv}(c') = \emptyset$ and hence easily concluded.

- $c = \exists x'.c'$: trivial.

□

Lemma 4.7 *For any multi-set of agents $t_1\{A_1\},\ldots,\ t_k\{A_k\}$ and any constraint c, if $A_1^{t_1}\ldots A_k^{t_k} \vdash_{\mathcal{C}^\dagger} c$ then there exists a derivation $(\emptyset;\mathbb{1};t_1\{A_1\},\ldots,t_k\{A_k\}) \xrightarrow{*} (\overline{x};\mathbb{c};!\Gamma)$ where $\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$ and $!\Gamma$ is a sequence of persistent asks , the variables $\overline{x}$ are free in c.*

Proof: Let us prove the result, by induction on the sequent $A_1^{t_1},\ldots,A_k^{t_k} \vdash_{\mathcal{C}^\dagger} c$ where the A_i's are agents and c a constraint. We shall consider without loss of generality, that in π the left introduction of $\forall$ and of $\multimap$ are always consecutive (if it is not the case, the rules can be permuted to obtain such a proof, see for example [7], noting that the *promotion* is the only case of unpermutability with $\forall$-*left* appears only in the constraint part, the right side of the sequent, and thus never bellow a $\multimap$-*right*).

First remark that this induction is meaningful. Indeed the only cuts which cannot be eliminated in an ILL proof deal with non-logical axioms, so they are of one of the following form:

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} c \quad \overline{c \vdash_{\mathcal{C}^\dagger} d}}{\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} d} \qquad \frac{\overline{c \vdash_{\mathcal{C}^\dagger} c'} \quad \Gamma^\dagger, c' \vdash_{\mathcal{C}^\dagger} d}{\Gamma^\dagger, c \vdash_{\mathcal{C}^\dagger} d}$$

Hence the application of the *cut* rule introduces sequents in which the new formula on the right is always a constraint. On the other hand the formulae on the left hand side remain sub-formulae of translation of agents.

One remarks also that $(A^t)^s = A^t$ and $s\{t\{A\}\} \longrightarrow A^t$ hence we can suppose without lost of generality that all A_i's are not of the form $t_i'\{A_i'\}$.

By induction on the proof π of $A_1^{t_1},\ldots,A_k^{t_k} \vdash_{\mathcal{C}^\dagger} c$:

- π is an *axiom*: $c \vdash_{\mathcal{C}^\dagger} d$. Since c is a constraint, Γ is of the form $t\{s : c'\}$ such that $c'^s = c$. Let $\mathbb{c} = \exists\overline{y}.(g|s{:}\overline{m})$ such that $\exists\overline{y}.(g\otimes\bigotimes\overline{m}) \dashv\vdash c'$ (we know it is possible thanks to the lemma 4.6). Then we have, by using the rule *tell*, $(\emptyset;\mathbb{1};t\{s : c'\}) \longrightarrow^* (\emptyset;\mathbb{c};\emptyset)$ and by using the lemma 4.2 $\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$.

- π ends with a cut:

$$\frac{\Gamma^\dagger \vdash c \quad \overline{c \vdash d}}{\Gamma^\dagger \vdash d} \qquad \text{or} \qquad \frac{\overline{c_1 \vdash c_2} \quad \Gamma^\dagger, c_2 \vdash d}{\Gamma^\dagger, c_1 \vdash d}$$

  The former case is immediate. In the latter there are two possible sub-cases the axiom is either of the form $c_1'^x \vdash_{\mathcal{C}^\dagger} c_2'^x$ such $c_1' \vdash_{\mathcal{C}} c_2'$ or of the form $x{=}y\otimes\dot{m}(x,\vec{t}) \vdash_{\mathcal{C}^\dagger} \dot{m}(y,\vec{t})$. By y induction hypothesis we know that $(\emptyset;\mathbb{1};\Gamma,c_2) \xrightarrow{*} (\overline{x};\mathbb{d};!\Gamma')$ such that $\exists\overline{x}.\mathbb{d}^\dagger \vdash_{\mathcal{C}^\dagger} d$. Just notice that the application of the *tell* rule that reduces the agent corresponding to c_2 can be applied on c_1 since $c_1' \vdash_{\mathcal{C}} c_2'$ and $\mathbb{c}'\otimes((x=y)|x{:}m(\vec{t})) \vdash c'\otimes(\emptyset|y{:}m(\vec{t}))$.

- π ends with **1**-left: note that $(\emptyset;\mathbb{1};t\{s{:}\mathbf{1}\},\mathbf{\Gamma}) \longrightarrow^* (\emptyset;\mathbb{1};\mathbf{\Gamma})$

- π ends with a $\otimes$-*left*:
$$\frac{\Gamma^\dagger, A \otimes B \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, A, B \vdash_{\mathcal{C}^\dagger} c \otimes d}$$
  - either $B \otimes B'$ is the translation a parallel composition of two agents, in such a case one can use the *parallel composition* rule.
  - or $B \otimes B'$ is the translation of a constraint of the form $t : (d \otimes d')$, in such a case just notice that $(\emptyset; \mathbb{1}; x\{t : (d \otimes d')\}, \Gamma)$ and $(\emptyset; \mathbb{1}; x\{t : d\}, x\{t : d'\}, \Gamma)$ have the same pseudo-successes.

- π ends with a $\otimes$-*right*:
$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} c \quad \Delta \vdash_{\mathcal{C}^\dagger} d}{\Gamma^\dagger, \Delta^\dagger \vdash_{\mathcal{C}^\dagger} c \otimes d}$$
  By induction hypothesis, we know there exists a derivation $(\emptyset; \mathbb{1}; \Gamma) \xrightarrow{*} (\overline{x}; \mathbb{c}; !\Gamma')$ and $(\emptyset; \mathbb{1}; \Delta) \xrightarrow{*} (\overline{y}; \mathbb{d}; !\Delta')$ such $\exists \overline{x}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$ and $\exists \overline{y}.\mathbb{d}^\dagger \vdash_{\mathcal{C}^\dagger} d$. Thanks to the monotonicity of $\xrightarrow{*}$ we can infer that $(\emptyset; \mathbb{1}; \Gamma, \Delta) \xrightarrow{*} (\overline{x}; \mathbb{c}; !\Gamma', \Delta) \xrightarrow{*} (\overline{x}, \overline{y}; \mathbb{c} \otimes \mathbb{d}; !\Gamma', !\Delta')$. To conclude we just notice that according to induction hypothesis, $\exists \overline{x}\, \overline{y}.(\mathbb{c} \otimes \mathbb{d})^\dagger \vdash_{\mathcal{C}^\dagger} c \otimes d$ if $\overline{x} \cap \mathrm{fv}(\mathbb{d}, d) \neq \emptyset$ and $\overline{y} \cap \mathrm{fv}(\mathbb{c}, c) \neq \emptyset$.

- π ends with $\exists$-right: immediate

- π ends with $\exists$-left:
$$\frac{\Gamma^\dagger, A^t \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, \exists x.A^t \vdash_{\mathcal{C}^\dagger} c}$$
  By induction hypothesis, we have $(\emptyset; \mathbb{1}; t\{A\}, \Gamma) \xrightarrow{*} (\overline{y}; \mathbb{c}; !\Gamma'))$ with $\exists \overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$. As we can suppose without lost of generality $x \notin \overline{y} \cap \mathrm{fv}\Gamma$ (since we work modulo α-conversation) and as $(\emptyset; \mathbb{1}; t\{\exists x.A\}, \Gamma) \xrightarrow{*} (x; \mathbb{1}; t\{A\}, \Gamma)$, by monotonicity of $\xrightarrow{*}$ we have
  $(\emptyset; \mathbb{1}; t\{\exists x.A\}, \Gamma) \xrightarrow{*} (x, \overline{y}; \mathbb{c}; !\Gamma')$. Because $x \notin \mathrm{fv}(c)$ and $\exists \overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$, we have $\exists x.\exists \overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$

- π ends with (thanks to the preliminary remarks on the permutability of rules):
$$\frac{\dfrac{\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} d^t[\vec{s}/\vec{z}] \qquad \Delta^\dagger, A^t[\vec{s}/\vec{z}] \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, \Delta^\dagger, d^t[\vec{s}/\vec{z}] \multimap A^t[\vec{s}/\vec{x}] \vdash_{\mathcal{C}^\dagger} c}}{\Gamma^\dagger, \Delta^\dagger, \forall \vec{z}.(d^t \multimap A^t) \vdash_{\mathcal{C}^\dagger} c}$$
  By induction hypothesis we have $(\emptyset; \mathbb{1}; \Gamma) \xrightarrow{*} (\overline{y}; \mathbb{d}; !\Gamma')$ such that $\exists \overline{y}.\mathbb{d}^\dagger \vdash_{\mathcal{C}^\dagger} d^t[\vec{s}/\vec{z}]$. By lemma 4.6 we know also there exists a constraint $\exists \overline{x}'.(g \otimes \bigotimes \overline{m})$ such that $\exists \overline{x}'.(g \otimes \bigotimes \overline{m}) \dashv\vdash d[\vec{s}/\vec{z}]$ and then by lemmas 4.2 and 4.5 we infer that $\mathbb{d} >_{\mathcal{C}} \exists \overline{x}'.(g|t{:}\overline{m})$. Thus by using the *monotonicity* of $\xrightarrow{*}$ and by applying the *ask* rule, one has $(\emptyset; \mathbb{1}; t\{\forall \vec{z}(c \to A)\}, !\Gamma) \xrightarrow{*} (\overline{x}; \mathbb{d}; t\{\forall \vec{z}(c \to A)\}, !\Gamma') \longrightarrow (\overline{x}; \mathbb{1}; \Gamma', t\{A[\vec{s}/\vec{z}]\})$. Moreover by induction

hypothesis, $(\emptyset;\mathbb{1};t\{A[\vec{s}/\vec{z}]\},\Delta) \xrightarrow{*} (\overline{y};\mathbb{c};!\Delta')$ with $\exists\overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$, hence by using once again the monotonicity of $\xrightarrow{*}$ we infer $(\emptyset;\mathbb{1};\Gamma,\Delta,t\{\forall\vec{z}(c \to A)\}) \xrightarrow{*} (\overline{x},\overline{y};\mathbb{c};!\Gamma',!\Delta')$. As $\exists\overline{y}''\overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$ if $\exists\overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$ and $\overline{y}'' \cap \mathrm{fv}(c) = \emptyset$ we can conclude.

- π ends with a dereliction. Thanks to the preliminary remarks on the permutability of rules there are only two sub-cases:

$$\frac{\Gamma^\dagger, d^t \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, !d^t \vdash_{\mathcal{C}^\dagger} c} \qquad \text{or} \qquad \frac{\Gamma^\dagger, \forall\vec{z}.(d^t \multimap A^t) \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, !\forall\vec{z}.(d^t \multimap A^t) \vdash_{\mathcal{C}^\dagger} c}$$

  In the former case, it is clear, just recall that $!c \vdash c$. In the latter by induction hypothesis, $(\emptyset;\mathbb{1};\Gamma,t\{\forall\vec{x}(d \to A)\}) \xrightarrow{*} (\overline{y};\mathbb{c};!\Gamma')$, with $\exists\overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$. Therefore by replacing in the previous derivation the *ask* rule that reduce the $t\{\forall\vec{x}(d \to A)\}$ agent (this reduction is necessary, otherwise $!\Gamma'$ would not be a sequence of persistent asks only), by the *persistent ask* rule we obtain $(\emptyset;\mathbb{1};t\{\forall\vec{x}(d \Rightarrow A)\},\Gamma) \xrightarrow{*} (\overline{y};\mathbb{c};\forall\vec{x}(d \Rightarrow A)\},!\Gamma')$. The result is then immediate.

- π ends with a promotion:

$$\frac{!\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} !c}{!\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} c}$$

  By induction hypothesis, $(\emptyset;\mathbb{1};!\Gamma) \xrightarrow{*} (\overline{y};\mathbb{c};!\Gamma')$ with $\exists\overline{y}.\mathbb{c}^\dagger \vdash_{\mathcal{C}^\dagger} c$. Just recall that $!c \vdash c$ to conclude.

- π ends with a weakening:

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, !d^t \vdash_{\mathcal{C}^\dagger} c} \qquad \text{or} \qquad \frac{\Gamma^\dagger \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, !\forall\overline{x}(d^t \multimap A^t) \vdash_{\mathcal{C}^\dagger} c}$$

  In the former case it is enough to notice that $(\emptyset;\mathbb{1};s\{t :!d\},\Gamma) \xrightarrow{*} (\emptyset;\mathbb{1};\Gamma)$ since $!d^t \vdash \mathbf{1}$. In the latter one just remarks that the addition of some persistent asks to a multi-set of an agent does not change its pseudo-successes.

- π ends with a contraction:

$$\frac{\Gamma^\dagger, !d^t, !d^t \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, !d^t \vdash_{\mathcal{C}^\dagger} c} \qquad \text{or}$$

$$\frac{\Gamma^\dagger, !\forall\overline{x}(d^t \multimap A^t), !\forall\overline{x}(d^t \multimap A^t) \vdash_{\mathcal{C}^\dagger} c}{\Gamma^\dagger, !\forall\overline{x}(d^t \multimap A^t) \vdash_{\mathcal{C}^\dagger} c}$$

  In the former, just note that for $\mathbb{d}$ such that $\mathbb{d}^\dagger \dashv\vdash !d^t$ (that is possible thanks to the lemma 4.6), we have $!d^t \vdash \mathbb{d}^\dagger \otimes \mathbb{d}^\dagger$. In the latter having two occurrences of the agent $\forall z(d \Rightarrow A)$ does not change anything, since all constraint consumed by two identical persistent asks can be consumed by only one.

□

Now, for a set S of constraint of $\mathcal{C}^\dagger$, let us note $\downarrow S = \{c \in \mathcal{C}^\dagger | \exists d \in S, d \vdash_{\mathcal{C}^\dagger} c\}$

Proposition 4.8 (Observation of pseudo-successes) *For every MLCC(C) agent A, we have:*

$$\downarrow (\mathcal{O}^{p\text{-}s}(A)^\dagger) = \{c \in \mathcal{C}^\dagger \mid A^\dagger \vdash_{\mathcal{C}^\dagger} c\}$$

Proof: One inclusion is obvious by applying the soundness theorem and by noting that for $!\Gamma, c \vdash c$. The other is a direct consequence of the previous lemma. □

Theorem 4.9 (Observation of stores) *For every MLCC(C) agent A, we have:*

$$\downarrow (\mathcal{O}^{store}(A)^\dagger) = \{c \in \mathcal{C}^\dagger \mid A^\dagger \vdash_{\mathcal{C}^\dagger} c \otimes \top\}$$

Proof: One inclusion is still obvious by applying the previous theorem 4.4 and by noting that $\Gamma, c \vdash c \otimes \top$. For the other inclusion use the previous proposition, above the right introduction of the tensor connective in $c \otimes \top$ and note that the property is preserved by all left introduction rules. □

Because our translation of MLCC agents implies the use of bangs (!) for the persistent asks, we are not able to exactly characterize final stores (and hence successes). Indeed the rule of weakening for the ! allows forgetting a formula corresponding to a persistent ask before it consumes any constraints it could. Nonetheless by supposing some properties over the constraints consumed by the persistent asks, we can characterize precisely an interesting subset of successes.

Definition 4.10 ($\mathcal{D}$-over agent) *An agent is $\mathcal{D}$-over if no guard c of its persistent asks belongs to $\mathcal{D}$.*

Definition 4.11 ($\mathcal{D}$-proof system) *We will say that $\mathcal{C}$ is a $\mathcal{D}$-proof constraint system, if for any constraint d of $\mathcal{D}$ and any constraint c of $\mathcal{C}$ we have:*

$$\text{if } d \vdash_{\mathcal{C}} c \otimes \top \text{ then } c \in \mathcal{D}$$

Theorem 4.12 (Observation of $\mathcal{D}$-success) *For any $\mathcal{D}$-over agent A, if $\mathcal{C}$ is a $\mathcal{D}$-proof system we have:*

$$\downarrow (\mathcal{O}^{\mathcal{D}\text{-}succ}(A)^\dagger) = \{d \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}^\dagger} d\}$$

Proof: One inclusion is obvious. Thank to the proposition 4.8, we know that for every constraint d of $\mathcal{D}$, there exists a derivation $(\emptyset; \mathbb{1}; x\{A\}) \xrightarrow{*} \kappa = (\overline{y}; \mathbb{d}; t_1\{\forall z_1(c_1 \Rightarrow A_1)\}, \ldots, t_k\{\forall z_k(c_k \Rightarrow A_k)\})$ such that $\exists y.\mathbb{d}^\dagger \vdash_{\mathcal{C}^\dagger} d$. Now to prove the other inclusion, we just need to prove that such a κ is irreducible.

First note that if $\mathcal{C}$ is $\mathcal{D}$-proof, then $\mathcal{C}^\dagger$ is $\mathcal{D}$-proof too. Then let us suppose that κ is reducible, in other word there exists a persistent ask $t_i\{\forall z_i(c_i \Rightarrow A_i)\}$ $(1 \leq i \leq k)$ in κ such that $\mathbb{d} >_{\mathcal{C}} \exists \overline{z}.(g|\overline{\mathbb{m}}, t_i : \overline{m})$ and $g \otimes \bigotimes \overline{m} \vdash_{\mathcal{C}} g' \otimes \bigotimes \overline{m'} \otimes c_i[\vec{s}/\vec{z_i}]$. Thanks to lemma 4.3, we have $\exists \overline{z}.(g \otimes \bigotimes \overline{\mathbb{m}}^\dagger \otimes m^{t_i}) \vdash_{\mathcal{C}^\dagger} c$ and then, since $\mathcal{C}^\dagger$ is $\mathcal{D}$-proof, we infer that $g \in \mathcal{D}$, $\mathbb{m} = \emptyset$ and $\overline{m} = \emptyset$. Hence we have $g \otimes \mathbf{1} \vdash_{\mathcal{C}} c_i[\vec{s}/\vec{z_i}] \otimes \top$ which contradicts the hypotheses, since $c_i \notin \mathcal{D}$, qed. □

5 A Module System for CLP

Through simple syntactical restrictions, the MLCC scheme presented above can be instantiated into a powerful yet simple module system for Constraint Logic Programming (CLP) languages. This resulting language, called mCLP for modular CLP, has been implemented in a "proof of concept" prototype available for download at the following address:
`http://contraintes.inria.fr/~haemmerl/pub/mclp.tgz`

5.1 mCLP Syntax

We adopt for mCLP a pragmatic syntax close to that of classical CLP systems. The syntax defined by the following grammar distinguishes declarations from goals as usual:

```
G ::= module(T,E){D} | T:p(S1,...,Sn) |
 p(S1,...,Sn) | c(S1,...,Sn) | G,G | G;G

D ::= p(S1,...,Sn) :- G.D | p(S1,...,Sn).D |
 :- G.D | ϵ
```

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 predicate symbols alphabet Σ_M.

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.

The goal $\mathtt{T:p(S}_1\mathtt{, \ldots, S}_n\mathtt{)}$ denotes the *external call* of the predicate `p/n` defined in the module `T`, which is distinguished from the *local call*, noted $\mathtt{T:p(S}_1\mathtt{, \ldots, S}_n\mathtt{)}$, of the predicate `p/n` defined in the current module.

5.2 Interpretation of mCLP into MLCC

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:

$$
\begin{array}{lcllcl}
[\![\mathtt{G}_1,\mathtt{G}_2]\!]^{\mathtt{T}} & = & [\![\mathtt{G}_1]\!]^{\mathtt{T}} \,||\, [\![\mathtt{G}_2]\!]^{\mathtt{T}} & [\![\mathtt{P}]\!]^{\mathtt{T}} & = & \mathtt{T:P} \\
[\![\mathtt{G}_1;\mathtt{G}_2]\!]^{\mathtt{T}} & = & [\![\mathtt{G}_1]\!]^{\mathtt{T}} + [\![\mathtt{G}_2]\!]^{\mathtt{T}} & [\![\mathtt{C}]\!]^{\mathtt{T}} & = & \mathtt{T:(!C)} \\
[\![\mathtt{module(S,E)\{D\}}]\!]^{\mathtt{T}} & = & \mathtt{S}\{[\![\mathtt{D}]\!]^{\mathtt{S}}_{\mathtt{E}}\} & [\![\mathtt{S:P}]\!]^{\mathtt{T}} & = & \mathtt{S:P}
\end{array}
$$

$$\begin{array}{lcl} [\![\texttt{:- G.D}]\!]^{\texttt{T}}_{\texttt{E}} & = & \exists \overline{\texttt{Y}} [\![\texttt{G}]\!]^{\texttt{S}} \,||\, [\![\texttt{D}]\!]^{\texttt{T}}_{\texttt{E}} \\ [\![\texttt{p}(\vec{\texttt{t}}) \texttt{.D}]\!]^{\texttt{T}}_{\texttt{E}} & = & \forall \vec{\texttt{X}} (\texttt{p}(\vec{\texttt{X}}) \Rightarrow \exists \overline{\texttt{Y}} [\![\vec{\texttt{X}} = \vec{\texttt{t}}]\!]^{\texttt{S}}) \,||\, [\![\texttt{D}]\!]^{\texttt{T}}_{\texttt{E}} \\ [\![\texttt{p}(\vec{\texttt{t}}) \texttt{ :- G.D}]\!]^{\texttt{T}}_{\texttt{E}} & = & \forall \vec{\texttt{X}} (\texttt{p}(\vec{\texttt{X}}) \Rightarrow \exists \overline{\texttt{Y}} [\![\vec{\texttt{X}} = \vec{\texttt{t}}, \texttt{G}]\!]^{\texttt{S}}) \,||\, [\![\texttt{D}]\!]^{\texttt{T}}_{\texttt{E}} \end{array}$$

where $\vec{\texttt{X}}$ is a set of fresh variables and $\overline{\texttt{Y}} = \text{fv}(\vec{\texttt{t}}, \texttt{G}) \setminus \texttt{E}$.

This translation is supposed to work on the linear constraint system $(\mathcal{CP}, \Vdash_{\mathcal{CP}})$ such that $\Vdash_{\mathcal{CP}}$ is the smallest set respecting the following conditions:

- If $(\texttt{C} \Vdash_{\mathcal{C}\circ} \texttt{C})$ then $(\texttt{C} \Vdash_{\mathcal{CP}} \texttt{C})$.
- For any predicate symbol p $\left(\texttt{p}(\vec{\texttt{X}}), \vec{\texttt{X}} = \vec{\texttt{Y}} \Vdash_{\mathcal{CP}} \texttt{p}(\vec{\texttt{Y}})\right)$.

where $\Vdash_{\mathcal{C}\circ}$ is the translation of the non-logical axioms of the classical constraint system $\mathcal{C}$ into linear logic (using for example the well know Girard's translation classical logical into linear logic [8]).

Notice that all the $[\![A]\!]^{\texttt{T}}_{\texttt{E}}$ are $\mathcal{C}$-over (see Def. 4.10) and that $\mathcal{CP}$ is $\mathcal{C}$-proof (see Def. 4.11), therefore all results of previous Section, and in particular theorem 4.12, can be applied to mCLP programs.

5.3 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(Iterative Deepening):

```
:-module(iter_deep, [Depth]){

   solve(G):-
      for(Depth,1,1000),
         write('Depth: '),
         write(Depth),
         nl,
         iterative_deepening(G,0).

   iterative_deepening(_,I) :-
      I >= Depth,
      !,
```

```
      fail.

   iterative_deepening(((A,B)),I) :-
      !,
      iterative_deepening(A,I),
      iterative_deepening(B,I).

   iterative_deepening(A,_) :-
      clause((A:-true)),
      !.

   iterative_deepening(A,I) :-
      clause((A:-B)),
      J is I+1,
      iterative_deepening(B,J).
}
```

5.4 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 known to the clauses inside the module definition, the corresponding implementation is accessible only from the clauses of the outside module.

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

Example(Quicksort):

```
:- module(sort, [Impl]){

   sort(List,SortedList):-
      Impl:quicksort(List, SortedList).

   :- module(Impl, []){

      quicksort([],[]).
      quicksort([X|Tail], Sorted) :-
         split(X, Tail, Small, Big),
         quicksort(Small, SortedSmall),
         quicksort(Big, SortedBig),
         list:append(SortedSmall,
            [X|SortedBig], Sorted).
```

```
      split(X, [],[],[]).
      split(X, [Y|Tail], [Y|Small], Big) :-
         X<Y, !,
         split(X, Tail, Small, Big).
      split(X, [Y|Tail], Small, [Y|Big]) :-
         split(X, Tail, Small, Big).
   }.
}.
```

The code protection property 3.6 ensures that no call to the `quicksort` predicate is possible outside the `sort` predicate.

5.5 Closures

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

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

Example:

```
:- module(iterator, []){

   forall([], _).
   forall([H|T], C) :- C:arg(H), forall(T, C).

   exists([H|_], C) :- C:arg(H).
   exists([_|T], C) :- exists(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:

Example(fd_ domain):

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

5.6 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(Parameterized quicksort):

```
:- module(sort, []){

   generic_sort(Sort, Order) :-
      module(Sort, [Order, Impl]){

         sort(List, SortedList):-
            Impl:quicksort(List, SortedList).

         :- module(Impl, [Order]){

            quicksort([],[]).
            quicksort([X|Tail], Sorted) :-
               split(X, Tail, Small, Big),
               quicksort(Small, SortedSmall),
               quicksort(Big, SortedBig),
               list:append(SortedSmall,
                  [X|SortedBig], Sorted).

            split(X, [],[],[]).
            split(X, [Y|Tail], [Y|Small], Big) :-
               Order:geq(X,Y), !,
               split(X, Tail, Small, Big).
            split(X, [Y|Tail], Small, [Y|Big]) :-
               split(X, Tail, Small, Big).
         }.
      }.
}.
```

6 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 for CLP supporting code hiding, closures and module parameterization.

We believe that this natural integration of module systems into programming languages is of a quite general scope for programming languages having logical variables.

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*, Lecture Notes in Computer Science. 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, New York, 1996.

[6] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, Feb. 2001.

[7] D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994.

[8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[9] R. Haemmerlé and F. Fages. Modules for Prolog revisited. Technical Report RR-5869, INRIA, 2006.

[10] L. Hallnäs. A proof-theoretic approach to logic programming. ii. programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, Oct. 1991.

[11] 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.

[12] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

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

[14] C. Palamidessi, V. Saraswat, F. Valencia, and B. Victor. On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In *Proc. of LICS'06*, pages 59–68. IEEE Computer Society Press, 2006.

[15] P. V. Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, Nov. 2003.

[16] P. Ruet and F. Fages. Concurrent constraint programming and mixed non-commutative linear logic. In *Proc. of Computer Science Logic CSL'97*, volume 1414 of *Lecture Notes in Computer Science*, pages 406–423. Springer-Verlag, Aug. 1997.

[17] V. A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993.

[18] V. A. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

[19] M. E. Stickel. A prolog technology theorem prover: implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 44:353–380, 1988.

A Intuitionistic Linear Logic

We give here a brief description of the intuitionistic version of Linear Logic (ILL) with its sequent calculus [8].

Definition A.1 (Formulae) *The* intuitionistic formulae *are built from atoms p, q, ... with the multiplicative connectives* $\otimes$ *(tensor) and* $\multimap$ *(linear implication), the additive connectives* & (with) *and* $\oplus$ (plus) *the exponential connective* ! *(bang), and the universal* $\forall$ *and existential* $\exists$ *quantifiers.*

Definition A.2 (Sequents) *The* intuitionistic sequents *are of the form* $\Gamma \vdash A$*, where* A *is a formula and* Γ *is a multi-set of formulae.*

The sequent calculus is given by the following rules, where the basic idea is that the disappearance of the weakening rule makes the conjunction $\otimes$ count the occurrences of formulae, and the implication $\multimap$ consume its premise:

Axiom - Cut

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

Constants

$$\frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A} \quad \vdash \mathbf{1} \quad \Gamma \vdash \top$$

$$\bot \vdash \quad \frac{\Gamma \vdash}{\Gamma \vdash \bot} \quad \Gamma, \mathbf{0} \vdash A$$

Multiplicatives

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

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

Additives

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}$$

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

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

Bang

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}$$

Quantifiers

$$\frac{\Gamma, A[t/x] \vdash B}{\Gamma, \forall x A \vdash B} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x A}\ x \notin \mathrm{fv}(\Gamma)$$

$$\frac{\Gamma, A \vdash B}{\Gamma, \exists x A \vdash B}\ x \notin \mathrm{fv}(y\Gamma, B) \qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}$$

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

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

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