Linear concurrent constraint programming: operational and phase semantics

François Fages[a], Paul Ruet[b], Sylvain Soliman[a]*

[a]INRIA Rocquencourt
BP 105
78153 Le Chesnay Cedex
France

[b]CNRS IML, Case 907,
163 avenue de Luminy,
13288 Marseille Cedex 9
France

Email: {Francois.Fages,Sylvain.Soliman}@inria.fr, ruet@iml.univ-mrs.fr
Tel.: +33 1 39 63 57 09 Fax: +33 1 39 63 54 69

March 2, 2000

Abstract

In this paper we give a logical semantics for the class CC of concurrent constraint programming languages and for its extension LCC based on linear constraint systems. Besides the characterization in intuitionistic logic of the stores of CC computations, we show that both the stores and the successes of LCC computations can be characterized in intuitionistic linear logic. We illustrate the usefulness of these results by showing with examples how the phase semantics of linear logic can be used to give simple "semantical" proofs of safety properties of LCC programs.

Contents

*This paper is an extended version of [7]. The main part of this work was done when the three authors were at CNRS, Ecole Normale Supérieure, Paris.

1 Introduction

Constraint programming is an emerging technology that has been proved very successful for complex system modeling and for solving declaratively combinatorial search problems [14]. The class CLP($\mathcal{X}$) of constraint logic programming languages, introduced by Jaffar and Lassez [13], extends logic programming by combining the pure Horn fragment of first-order logic (FOL) for defining relations, with a fixed decidable FOL language of primitive constraints over some structure $\mathcal{X}$. The logical semantics of logic programming for both successes and finite failures extend smoothly to CLP languages, by considering the consequences in classical logic of both the program and the theory of the structure $th(\mathcal{X})$. This is achieved in pure CLP for various observable properties of the execution: the existence of a successful derivation to a query [13], the set of computed answer constraints [21, 8], finite failures [13], the set of computed constraints with constructive negation [37, 6], etc. For example, computed answer constraints (i.e. final states of computations) can be observed logically: any computed constraint entails the initial goal (modulo the logical translation of the program $P^\star$ and the constraint system $\mathcal{C}$); conversely any constraint c entailing a goal G is covered (again modulo $P^\star$ and $\mathcal{C}$) by a finite set of computed constraints $c_1 \dots c_n$, i.e. $\mathcal{C} \vdash \forall(c_1 \dots c_n \Rightarrow c)$. This allows for powerful analysis methods and simple program transformation techniques based on logical equivalences.

The class CC($\mathcal{X}$) of *Concurrent Constraint* programming languages introduced by Saraswat [32] in 1987 arose as a natural combination of constraint logic programming and concurrent logic programming, with the introduction of a synchronization mechanism based on constraint entailment [21]. CC programming is a model of concurrent computation, where agents communicate through a shared store, represented by a constraint, which expresses some *partial information* on the values of the variables involved in the computation. An agent may add information c to the store (agent $tell(c)$), or ask the store to entail a given constraint ($c \rightarrow A$). Communication is *asynchronous*: agents can remain idle, and senders (constraints c) are not blocking. The synchronization mechanism of CC languages gives an account for the co-routining facilities of implemented CLP systems, like the freeze predicate of Prolog, the delay mechanism of CLP($\mathcal{R}$) [13], or the constraint propagation schemes of CLP(FD). It also opens, to some extent, constraint programming to a new field of applications which are traditional in concurrent programming, like reactive systems and protocol specifications.

From the logic programming tradition however, the operational aspects of CC programming should also be closely connected to a logical semantics for reasoning about programs at different levels of abstraction, getting rid of useless details of the execution. The monotonic evolution of the store during CC computations provides CC languages with a simple denotational semantics in which agents are identified to closure operators on the semi-lattice of constraints [35, 15]. Such denotational semantics are used in [5] to obtain a complete calculus for partial correctness assertions where the rules of the proof system mirror the equations of the denotational semantics.

In this article, we explore another route based on Girard's intuitionistic linear logic (ILL) [10]. We review the semantics of CC languages in the logic programming paradigm based on linear logic and we investigate the use of the phase semantics for proving safety properties of CC programs.

Outline of the paper.

Section 2 presents a natural extension of CC languages in this context, namely Linear CC (LCC) where the constraint system is axiomatized in linear logic. LCC is an extension of CC, somewhat similar to [3] or [33], but where constraints are *consumed* by ask agents without dependency maintenance or recomputation. Linear constraint systems have also been proposed in [34] in a higher-order setting which will not be considered here. From an operational point of view, LCC extends CC in a fundamental way by introducing some forms of imperative programming, particularly useful for reactive systems. Standard CC programs can however be recovered by the usual translation of intuitionistic logic into linear logic [10].

Section 3 settles the basic soundness and completeness results of CC and LCC operational semantics w.r.t. intuitionistic linear logic, relying on [30] and preliminary results from [29]. Results similar to those of this section are part of the folklore on CC languages [19, 34] but have not been published. Here we prove that 1) the stores of CC computations can be characterized in intuitionistic logic, and 2) both the stores and the successes of CC and LCC computations can be characterized in intuitionistic linear logic. Completeness results show that ILL can be used to prove *liveness properties* of LCC programs, i.e. properties expressing that something good will eventually happen. This is developed for both "may" and "must" properties.

Then we show in section 4.2 how *safety properties* of CC and LCC programs (i.e. that some derivations never happen) can be proved using the phase semantics of linear logic. The method relies on the soundness theorem of LCC computations w.r.t. linear logic, and on the soundness theorem of linear logic w.r.t. the phase semantics. Completeness results simply say in this context that for various classes of observable properties of the program, if the property holds then such a "phase semantical proof" exists. The method is illustrated with several examples of LCC programs for protocol specifications.

Related work.

The connection between CC languages and linear logic is based on the logic programming paradigm in a broad sense, that identifies programs-as-formulas and execution-as-proof-search. This paradigm was applied to linear logic with the notion of uniform proofs [23, 12] and focusing proofs [1], and further works on the design of concurrent languages based on proof search in LL [16, 27].

However our approach is analytical in that we study an existing programming language CC, and model CC computations in a fragment of LL. On the other hand we model properties of infinite CC computations through the observation of accessible stores which has no counterpart in the uniform proofs approach. Therefore both series of results are quite different.

In [22] it is shown how intuitionistic logic and linear logic can be used to encode transition systems in various ways. Particularly interesting is the augmentation of these logics with a proof theoretical treatment of definitions that makes it possible to reason about both "may" properties and "must" properties in a symetrical way within a single translation of programs.

We shall not adopt this approach however, in order to stay in the framework of intuitionistic and linear theories.

Recently, phase semantics has gained interest in its applications to cut elimination [25], complexity of provability and decidability [17] (see [18] for a survey). Section 4.2 presents a new field of application of the phase semantics, yet unexplored though quite natural in the paradigm of concurrent logic programming.

2 CC operational semantics

In this paper, a set of variables is denoted by X, Y,..., the set of free variables occurring in a formula A is denoted by $\mathrm{fv}(A)$, a sequence of variables is denoted by $\vec{x}$, $A[\vec{t}/\vec{x}]$ 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 for avoiding variable clashes).

For a set S, $S^\star$ denotes the set of finite sequences of elements in S. For a transition relation $\rightarrow$, $\rightarrow^\star$ denotes the transitive and reflexive closure of $\rightarrow$.

2.1 CC

Definition 2.1 (Intuitionistic constraint system) *A* constraint system *is a pair* $(\mathcal{C}, \vdash_\mathcal{C})$, *where:*

- $\mathcal{C}$ *is a set of formulas (the* constraints*) built from a set* V *of variables, a set* Σ *of function and relation symbols, with logical operators:* 1 *(true), the conjunction* $\wedge$ *and the existential quantifier* $\exists$*;* $\mathcal{C}$ *is assumed to be closed by renaming, conjunction and existential quantification;*
- $\Vdash_\mathcal{C}$ *is a subset of* $\mathcal{C} \times \mathcal{C}$*, which defines the non-logical axioms of the constraint system. Instead of* $(c, d) \in \Vdash_\mathcal{C}$*, we write* $c \Vdash_\mathcal{C} d$*.*
- $\vdash_\mathcal{C}$ *is the least subset of* $\mathcal{C}^\star \times \mathcal{C}$ *containing* $\Vdash_\mathcal{C}$ *and closed by the rules of intuitionistic logic (IL) for* 1*,* $\wedge$ *and* $\exists$*:*

$$\Gamma, c \vdash c \qquad \frac{\Gamma, c \vdash d \qquad \Gamma \vdash c}{\Gamma \vdash d} \qquad \vdash 1 \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

$$\frac{\Gamma, d, d \vdash c}{\Gamma, d \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma, d \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma \vdash \exists x c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, \exists x c \vdash d}\ x \notin fv(\Gamma, d)$$

$$\frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1 \wedge c_2} \qquad \frac{\Gamma, c_1 \vdash c}{\Gamma, c_1 \wedge c_2 \vdash c} \qquad \frac{\Gamma, c_2 \vdash c}{\Gamma, c_1 \wedge c_2 \vdash c}$$

In the following, $c, d, e \ldots$ will denote constraints. Note that the intuitionistic logical framework (rather than the classical one) is not essential, it is simply sufficient, taking into account that the constraints are only built from conjunctions and existential quantifications.

Definition 2.2 (Agents) *The syntax of CC* agents *is given by the following grammar:*

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

where $\parallel$ *stands for parallel composition,* $+$ *for non-deterministic choice,* $\exists$ *for variable hiding and* $\rightarrow$ *for blocking ask. The atomic agents* $p(\vec{x}) \ldots$ *are called* process calls *or* procedure calls*, we assume that the arguments in the sequence* $\vec{x}$ *are all distinct variables.*

Traditionally, the *ask* agents in CC are not written with a universal quantifier [32]. The reason is perhaps that in the Herbrand domain, a CC($\mathcal{H}$) ask agent, like for example $\forall y, z(x = [y,z] \rightarrow A(x,y,z))$ for decomposing a list x, can be written without a universal quantifier, by duplicating the constraint in the guard and in the body of the *ask*: $(\exists y, z\ x = [y|z]) \rightarrow \exists y, z\ (tell(x = [y|z]) \parallel A(x,y,z))$. This programming trick is rather cumbersome however and does not generalize to every constraint domain nor to linear constraint systems. Therefore we shall not adopt it in this paper, and we shall make explicit the universal quantification of variables in *ask* agents.

Recursion is obtained by declarations:

Definition 2.3 (Declarations) *The syntax of* declarations *is given by the following grammar:*

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

The set of declarations of a CC program, denoted by $\mathcal{D}$, is the closure by variable renaming of a set of declarations given for distinct procedure names p.

Definition 2.4 (Programs) *A* program *$D.A$ is a declaration D together with an initial agent A.*

CC programs are parameterized by a constraint system. In general the constraint system $\mathcal{C}$ will be implicit in our presentation, both in the transition system and the constraint entailment relation. Similarly the set of declarations $\mathcal{D}$ will be kept implicit.

We make the usual hypothesis that in a declaration $p(\vec{x}) = A$, all the free variables occurring in A occur in $\vec{x}$. Notice that this is exactly the meaning associated with the Horn clauses in the logic programming languages: the local variables in a clause, that are free in the body but have no occurrence in the head, are considered (implicitly in the syntax, explicitly in the semantics) as existentially quantified in the body (because they are universally quantified in the clause).

For example, a Prolog-like program for concatenating two lists $L1$ and $L2$ in $L3$ will be written with the CC($\mathcal{H}$) declaration (over the Herbrand constraint system)

$$\begin{aligned} append(L1, L2, L3) = \ & (tell(L1 = []) \parallel tell(L2 = L3)) + \\ & \exists E, L, R(tell(L1 = [E|L]) \parallel tell(L3 = [E|R]) \parallel append(L, L2, R)) \end{aligned}$$

while a directional program for concatening two input lists $L1$ and $L2$ into an output list $L3$ will be written in CC($\mathcal{H}$) with the declaration

$$\begin{aligned} app(L1, L2, L3) = \ & (L1 = [] \rightarrow tell(L2 = L3)) + \\ & \forall E, L(L1 = [E|L] \rightarrow \exists R(tell(L3 = [E|R]) \parallel app(L, L2, R))) \end{aligned}$$

The operational semantics is defined on configurations (rather than agents) where the store is distinguished from agents:

Definition 2.5 (Configurations) *A* configuration *is a triple $(X; c; A)$, where c is a constraint called the store, A is an agent or $\emptyset$ if empty, and X is a set of variables, called the* hidden *variables of c and A.*

The operational semantics is defined by a transition system which does not take into account specific evaluation strategies. The transitions system is given in the style of the CHAM [2] (see also [28]). We thus distinguish a congruence relation between syntactic elements from the very transition relation between configurations.

Definition 2.6 (Congruence) *The* structural congruence $\equiv$ *is the least congruence satisfying the rules of table 1.*

$\mathcal{C}$-Equivalence	$\dfrac{c \dashv\vdash_{\mathcal{C}} d}{c \equiv d}$
α-Conversion	$\dfrac{z \notin fv(A)}{\exists y A \equiv \exists z A[z/y]}$
Parallel composition	$A \parallel B \equiv B \parallel A$ $A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$

Table 1: Structural congruence

The axioms of associativity and commutativity for parallel composition provide agents with a structure of multiset. From now on, by a slight abuse of notation, we will identify the agent of a configuration to the multiset of its subagents in parallel composition. We will write $\Gamma, \Delta, ...$ for multisets of agents in configurations. Congruence is extended to multisets of agents in the obvious way: $\Gamma \equiv \Gamma'$ iff $\Gamma = \{A_1, \ldots, A_n\}$, $\Gamma' = \{A'_1, \ldots, A'_n\}$ and $\forall i = 1, \ldots, n$, $A_i \equiv A'_i$. Two configurations are said *congruent*, $(X; c; \Gamma) \equiv (X'; c'; \Gamma')$, when the sets X and X' are equal, the constraints c and c' are $\mathcal{C}$-equivalent, and the multisets of agents Γ and Γ' are congruent.

Definition 2.7 (Transitions) *The* transition *relation $\longrightarrow_{CC}$ is the least transitive relation on configurations satisfying the rules of table 2.*

Equivalence	$\dfrac{(X; c; \Gamma) \equiv (X'; c'; \Gamma') \longrightarrow_{CC} (Y'; d'; \Delta') \equiv (Y; d; \Delta)}{(X; c; \Gamma) \longrightarrow_{CC} (Y; d; \Delta)}$
Tell	$(X; c; tell(d), \Gamma) \longrightarrow_{CC} (X; c \wedge d; \Gamma)$
Ask	$\dfrac{c \vdash_{\mathcal{C}} d[\vec{t}/\vec{y}]}{(X; c; \forall \vec{y}(d \to A), \Gamma) \longrightarrow_{CC} (X; c; A[\vec{t}/\vec{y}], \Gamma)}$
Hiding	$\dfrac{y \notin X \cup fv(c, \Gamma)}{(X; c; \exists y A, \Gamma) \longrightarrow_{CC} (X \cup \{y\}; c; A, \Gamma)}$
Procedure calls	$\dfrac{(p(\vec{y}) = A) \in D}{(X; c; p(\vec{y}), \Gamma) \longrightarrow_{CC} (X; c; A, \Gamma)}$
Blind choice	$(X; c; A + B, \Gamma) \longrightarrow_{CC} (X; c; A, \Gamma)$ $(X; c; A + B, \Gamma) \longrightarrow_{CC} (X; c; B, \Gamma)$

Table 2: CC transition relation

In this presentation of the transition relation it is clear that the set of hidden variables in configurations can only grow along a derivation:

Proposition 2.8 *If* $(X;c;\Gamma) \longrightarrow_{CC} (Y;d;\Delta)$ *then* $X \subseteq Y$.

The agents and declarations without $+$ and $\forall$ are called *deterministic*. This name is justified by the following proposition:

Proposition 2.9 (Confluence [35]) *For any deterministic configuration* κ *with deterministic declarations, if* $\kappa \longrightarrow_{CC} \kappa_1$ *and* $\kappa \longrightarrow_{CC} \kappa_2$*, then there exists a deterministic configuration* κ' *such that* $\kappa_1 \longrightarrow^*_{CC} \kappa'$ *and* $\kappa_2 \longrightarrow^*_{CC} \kappa'$.

Another property of CC programs is that the execution is *extensive* (constraints are only added to the store during execution) and *monotonic*:

Proposition 2.10 (Extensivity [35])
If $(X;c;\Gamma) \longrightarrow^*_{CC} (Y;d;\Delta)$ *then* $\exists Y d \vdash_{\mathcal{C}} \exists X c$.

Proposition 2.11 (Monotonicity [35])
If $(X;c;\Gamma) \longrightarrow^*_{CC} (Y;d;\Delta)$*, then for every multiset of agents* Σ *and every constraint* e *with* $fv(e,\Sigma) \cap (Y \cup fv(d,\Delta) \subseteq X \cup fv(c,\Gamma)$*,* $(X;c \wedge e;\Gamma,\Sigma) \longrightarrow^*_{CC} (Y;d \wedge e;\Delta,\Sigma)$.

As usual, the precise operational semantics depends on the choice of observables. We shall consider accessible stores, success stores and suspensions:

Definition 2.12 (Observables) *The* store *of a configuration* $(X;c;\Gamma)$ *is the constraint* $\exists X c$. *We will say that* $\exists X d$ *is an* accessible store *from the agent* A *and the initial store* c*, if there exists a multiset of formulas* Γ *such that* $(\emptyset;c;A) \longrightarrow^*_{CC} (X;d;\Gamma)$.

A success store *(resp. a* success*) for an agent* A *and an initial store* c *is a constraint* $\exists X d$ *(resp. a configuration* $(X;d;\emptyset)$*) such that* $(\emptyset;c;A) \longrightarrow^*_{CC} (X;d;\emptyset)$.

A terminal configuration *for* A *and initial store* c *is a configuration* $(X;d;d_1 \rightarrow A_1,\dots,d_n \rightarrow A_n)$ *such that* $n \geq 0$*,* $(\emptyset;c;A) \longrightarrow^*_{CC} (X;d;d_1 \rightarrow A_1,\dots,d_n \rightarrow A_n)$ *and for no* i*,* $d \vdash_{\mathcal{C}} d_i$*. The store* $\exists X d$ *is called a* suspension *if the configuration isn't a success (i.e.* $n > 0$*).*

It is easy to see that, by the monotonicity and extensivity properties of CC programs, the operational behavior of CC programs under these observables is completely characterized by their behavior on agents with an empty initial store. Namely the accessible stores from A with initial store c are the conjunctions of c and of the accessible stores from $A \parallel (tell\ c)$ with the empty initial store (prop. 2.11), the success stores from A with c are the success stores of $A \parallel tell(c)$ with the empty initial store, and similarly for suspensions. Therefore the operational semantics can be defined with the empty initial store (i.e. the constraint true noted 1):

Definition 2.13 (Operational semantics)

$\mathcal{O}^{store}_{CC}(\mathcal{C},\mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid$ *for some configuration* Γ*,* $(\emptyset;1;A) \longrightarrow^*_{CC} (X;d;\Gamma)\}$
$\mathcal{O}^{term}_{CC}(\mathcal{C},\mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid$ *for some* Γ*,* $(\emptyset;1;A) \longrightarrow^*_{CC} (X;d;\Gamma) \not\longrightarrow_{CC}\}$
$\mathcal{O}^{success}_{CC}(\mathcal{C},\mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid (\emptyset;1;A) \longrightarrow^*_{CC} (X;d;\emptyset)\}$

Remark on non-determinism.

In the transition system, we have adopted the *blind-choice* rule: the non-deterministic agent $A + B$ can behave either like A or like B. Replacing the blind choice rule (also called "internal choice") by the rules for the *one-step guarded choice* (also called "external choice"):

$$\frac{(X;c;A,\Gamma) \longrightarrow_{CC} (Y;d;\Delta)}{(X;c;A+B,\Gamma) \longrightarrow_{CC} (Y;d;\Delta)} \quad \text{and} \quad \frac{(X;c;B,\Gamma) \longrightarrow_{CC} (Y;d;\Delta)}{(X;c;A+B,\Gamma) \longrightarrow_{CC} (Y;d;\Delta)} \quad .$$

would obviously change the suspensions of a program. For instance, let $\kappa = (\emptyset; c; (c \to tell(1)) + (d \to tell(1)))$ with $c \not\vdash_{\mathcal{C}} d$, then for the blind choice, κ has two possible derivations: $\kappa \longrightarrow_{CC} (\emptyset; c; c \to tell(1)) \longrightarrow_{CC} (\emptyset; c; tell(1)) \longrightarrow_{CC} (\emptyset; c; \emptyset)$ and $\kappa \longrightarrow_{CC} (\emptyset; c; d \to tell(1)) \not\longrightarrow_{CC}$, whereas the second derivation is not accepted by the one-step choice. It is worth noting however that the set of successes, as well as the set of accessible stores remain the same under both interpretations. For the sake of simplicity we have not explicitly treated here all forms of non-determinism, but the results presented in this paper remain valid as long as only accessible stores and successes are observed.

Proposition 2.14 *Let $\mathcal{O}^{store}_{CC'}$ and $\mathcal{O}^{success}_{CC'}$ denote the operational semantics defined above with the one-step guarded choice rules instead of the blind choice rules. Let A be a CC agent, we have:*
$\mathcal{O}^{store}_{CC}(\mathcal{C}, \mathcal{D}.A) = \mathcal{O}^{store}_{CC'}(\mathcal{C}, \mathcal{D}.A)$ *and* $\mathcal{O}^{success}_{CC}(\mathcal{C}, \mathcal{D}.A) = \mathcal{O}^{success}_{CC'}(\mathcal{C}, \mathcal{D}.A)$

Proof. Obvious induction: let $\longrightarrow_{CC'}$ be the transition relation with the one-step guarded choice rules, consider a $\longrightarrow_{CC}$ derivation, it diverges from a $\longrightarrow_{CC'}$ derivation when it stops after a transition with the blind choice rule, but then: 1) the observed store has not changed, and 2) the terminal configuration is not a success. □

The monotonicity and extensivity properties provide CC with a denotational semantics, where the agents are seen as closure operators on the semi-lattice of constraints [35, 15]. In this paper however, we shall also be concerned with a variant of CC languages where constraints are formulas in linear logic [10] and where extensivity is dropped.

2.2 Linear CC

Roughly speaking, there are two reasons to consider linear constraints:

– on one hand, as we shall see in section 3.2, linear logic enables the characterization of finer observables than intuitionistic logic, and is therefore a natural semantics for CC;

– on the other hand, variants of CC, where the constraints can be consumed by ask agents and thus removed from the store, have been introduced by Saraswat and Lincoln [34], then further studied in [3, 38]: these variants enhance significantly the expressive power of CC (see the examples of communication protocol programs in section 2.2.3) and the constraints are naturally modeled as formulas of linear logic.

In this section we present such a version, LCC, and give a translation from CC into LCC respecting the transition system, so that LCC is a refinement of CC, and the logical characterization that we will make on the operational behavior of LCC is also correct for CC.

2.2.1 Syntax

As for CC, we define the constraint systems, the agents, the configurations and the transition system. The essential difference with CC is that constraints are formulas of linear logic and that communication (the *ask* rule) consumes information.

Definition 2.15 (Linear constraint system) *A* linear constraint system *is a pair* $(\mathcal{C}, \vdash_{\mathcal{C}})$, *where:*

- $\mathcal{C}$ *is a set of formulas (the* linear constraints*) built from a set* V *of variables, a set* Σ *of function and relation symbols, with logical operators: the multiplicative conjunction* $\otimes$*, its neutral element* 1*, the existential quantifier* $\exists$*, the exponential connective* $!$ *and the constant* $\top$*;*

- $\Vdash_{\mathcal{C}}$ *is a subset of* $\mathcal{C} \times \mathcal{C}$ *which defines the non-logical axioms of the constraint system.*

- $\vdash_{\mathcal{C}}$ *is the least subset of* $\mathcal{C}^{\star} \times \mathcal{C}$ *containing* $\Vdash_{\mathcal{C}}$ *and closed by the following rules (fv(A) denotes the set of free variables occurring in A):*

$$c \vdash c \qquad \frac{\Gamma, c \vdash d \qquad \Delta \vdash c}{\Gamma, \Delta \vdash d} \qquad \vdash 1 \qquad \Gamma \vdash \top \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

$$\frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \qquad \frac{\Gamma \vdash c_1 \qquad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \qquad \frac{\Gamma \vdash c[t/x]}{\Gamma \vdash \exists x c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, \exists x c \vdash d}\ x \notin fv(\Gamma, d)$$

$$\frac{\Gamma, c \vdash d}{\Gamma, !c \vdash d} \qquad \frac{!\Gamma \vdash d}{!\Gamma \vdash !d} \qquad \frac{\Gamma \vdash d}{\Gamma, !c \vdash d} \qquad \frac{\Gamma, !c, !c \vdash d}{\Gamma, !c \vdash d}$$

These are the rules of intuitionistic linear logic (ILL) for 1, $\otimes$, $\exists$ and $!$ (see appendix A).

Note that the intuitionistic constraint systems of the previous section can be recovered by writing all constraints under a $!$, as in the usual translation of intuitionistic logic into linear logic [10]. We have chosen to limit the use of $!$ to constraints only, because the usual replication operator of process calculi (like the π-calculus [24], where it is also noted $!$) does not have the same behavior as the exponential connective: it allows replication ($!A \longrightarrow_{LCC} (!A \parallel !A)$) but not erasing ($!A \not\longrightarrow_{LCC} \emptyset$).

In general, linear constraints are not erasable in the sense that $d \not\vdash 1$. One defines the following *subsumption* preorder between linear constraints:

Definition 2.16 *The* subsumption preorder $>$ *is defined by:* $c > d$ *iff* $c \vdash_{\mathcal{C}} d \otimes \top$.

Recall that $\top$ is the additive true constant which is neutral for $\&$ (see the appendix), the axiom for $\top$ is $\Gamma \vdash \top$, hence $c > d$ iff there exists some formula A s.t. $c \vdash_{\mathcal{C}} d \otimes A$.

The definition of LCC *agents*, *declarations* and *configurations* is the same as in CC (section 2.1), and we assume again very naturally that in a declaration $p(\vec{x}) = A$, all the free variables occurring in A have a free occurrence in $p(\vec{x})$, and that in a relation $c \Vdash_{\mathcal{C}} d$, all the free variables occurring in d have a free occurrence in c.

Definition 2.17 *The* structural congruence $\equiv$ *is the same as for CC (Definition 2.7). The* transition *relation* $\longrightarrow_{LCC}$ *is defined by the same rules as for CC (Definition 2.7), except for* **LinearTell** *and* **LinearAsk** *which replace* **Tell** *and* **Ask***:*

LinearTell	$(X; c; tell(d), \Gamma) \longrightarrow_{LCC} (X; c \otimes d; \Gamma)$
LinearAsk	$\dfrac{c \vdash_{\mathcal{C}} d \otimes e[\vec{t}/\vec{y}]}{(X; c; \forall \vec{y}(e \rightarrow A), \Gamma) \longrightarrow_{LCC} (X; d; A[\vec{t}/\vec{y}], \Gamma)}$

The calculus is intrinsically non-deterministic, even without the choice operator + nor the $\forall$, since several constraints can satisfy the condition of the **LinearAsk** rule with different residual stores for d. Of particular interest in this context are the *synchronization constraints* which are roughly linear atomic constraints without non logical axioms [31]. An *ask* with a synchronization constraint without $\forall$ amounts to simple multiset manipulations and is thus deterministic. We shall see examples of this in section 2.2.3 with communication protocol programs.

Because constraints are linear formulas, we must slightly modify the definition of terminal stores and suspensions.

Definition 2.18 (Observables) *The* successes *are defined as in CC.*

A store *accessible from A and the initial store c is a constraint e such that there exist a constraint d and a multiset Γ of agents such that $(\emptyset; c; A) \longrightarrow^*_{LCC} (\emptyset; d; \Gamma)$ and $d > e$.*

A terminal store *for A with initial store c is a configuration $(X; d; d_1 \to A_1, \ldots, d_n \to A_n)$ such that $n \geq 0$, $(\emptyset; c; A) \longrightarrow^*_{LCC} (X; d; d_1 \to A_1, \ldots, d_n \to A_n)$ and for no i, $d > d_i$. The store $\exists X d$ is called a* suspension *if the configuration isn't a success (i.e. if $n > 0$).*

An agent A with initial store c suspends with the store d on the constraints $d_1, ..., d_n$, *if there exist a suspension for A and c of the form $(X; d; d_1 \to A_1, \ldots, d_n \to A_n)$.*

One major breakthrough in the expressive power of LCC is due to the loss of extensivity, i.e. the ability to express by LCC agents non monotonic evolutions of the store, where constraints can be added and then consumed by linear ask operations. It is worth noting however that the monotonicity of transitions is still preserved in LCC:

Proposition 2.19 (Monotonicity)
If $(X; c; \Gamma) \longrightarrow^*_{LCC} (Y; d; \Delta)$, then for every multiset of agents Σ and every constraint e with $fv(e, \Sigma) \cap (Y \cup fv(d, \Delta) \subseteq X \cup fv(c, \Gamma)$, $(X; c \otimes e; \Gamma, \Sigma) \longrightarrow_{LCC} (Y; d \otimes e; \Delta, \Sigma)$.

As in the previous section, the observable properties of LCC computations from an empty initial store suffice to recover the properties of LCC computations from an arbitrary initial store. The argument is the same for the observation of successes and terminal stores, but is somewhat more tricky for the observation of accessible stores[1]:

Proposition 2.20 *Let $\mathcal{C}$ be a constraint system, and $\mathcal{C}'$ be the constraint system obtained by adding a new constraint token d to $\mathcal{C}$. The set of accessible stores from a configuration $(\emptyset; c; A)$ in $\mathcal{C}$ is the set $\{e \in \mathcal{C} \mid e \otimes d \in \mathcal{O}^{store}_{LCC}(\mathcal{C}', \mathcal{D}.(tell(c \otimes d) \parallel (d \to A)))\}$.*

Proof. As d is a new constraint token, the only possible transitions are:
$(\emptyset; 1; tell(c \otimes d \otimes d) \parallel (d \to A)) \longrightarrow_{LCC} (\emptyset; c \otimes d \otimes d; d \to A) \longrightarrow_{LCC} (\emptyset; c \otimes d; A) \longrightarrow_{LCC} \cdots$.

The stores accessible from $(\emptyset; c; A)$ are thus the stores $e \in \mathcal{C}$ (i.e. not containing d) such that $e \otimes d$ is an accessble store from $(\emptyset; 1; tell(c \otimes d \otimes d) \parallel (d \to A))$[2]. $\square$

Definition 2.21 (Operational semantics)

$\mathcal{O}^{store}_{LCC}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid \textit{for some configuration } \Gamma,\ (\emptyset; 1; A) \longrightarrow^*_{LCC} (X; d; \Gamma)\}$

$\mathcal{O}^{term}_{LCC}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid \textit{for some } \Gamma,\ (\emptyset; 1; A) \longrightarrow^*_{LCC} (X; d; \Gamma) \not\longrightarrow_{LCC}\}$

$\mathcal{O}^{success}_{LCC}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid (\emptyset; 1; A) \longrightarrow^*_{LCC} (X; d; \emptyset)\}$

[1] As an alternative, we could have defined the operational semantics of LCC programs with arbitrary initial stores, the generalization of the results is straightforward.

[2] Note that an encoding with $c \otimes d$ instead of $c \otimes d \otimes d$ would not allow us to decide whether 1 is an accessible store or not.

2.2.2 Translation from CC to LCC

The LCC languages are a refinement of usual CC languages. Indeed the extensivity of CC can simply be restored with the exponential connective ! of linear logic, allowing replication of hypotheses and thus avoiding constraint consumption during an application of the *ask* rule:

Definition 2.22 *Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a constraint system. We define the* translation *of $(\mathcal{C}, \Vdash_{\mathcal{C}})$, which is the linear constraint system $(\mathcal{C}^\circ, \Vdash_{\mathcal{C}}^\circ)$, as follows, at the same time as the translation of CC agents to LCC agents:*

$$\begin{array}{ll} c^\circ = !c, \text{ if } c \text{ is an atomic constraint} & \\ (c \wedge d)^\circ = c^\circ \otimes d^\circ & (\exists x c)^\circ = \exists x c^\circ \\ tell(c)^\circ = tell(c^\circ) & p(\vec{x})^\circ = p(\vec{x}) \\ (A \parallel B)^\circ = A^\circ \parallel B^\circ & (A + B)^\circ = A^\circ + B^\circ \\ (\forall \vec{x}(c \to A))^\circ = \forall \vec{x}(c^\circ \to A^\circ) & (\exists x A)^\circ = \exists x A^\circ \end{array}$$

$\mathcal{C}^\circ = \{c^\circ \mid c \in \mathcal{C}\}$.
The entailment relation $\Vdash_{\mathcal{C}}^\circ$ is defined by: $c \Vdash_{\mathcal{C}} d$ iff $c^\circ \Vdash_{\mathcal{C}}^\circ d^\circ$.
The relation $\vdash_{\mathcal{C}}{}^\circ$ is obtained from $\Vdash_{\mathcal{C}}{}^\circ$ by the rules of linear logic for 1, $!$, $\otimes$ and $\exists$.
The translation of a CC configuration $(X; c; \Gamma)$ is the LCC configuration $(X; c^\circ; \Gamma^\circ)$.

For constraints, the above translation is a well-known translation of intuitionistic logic into linear logic [10, p.81], hence:

Proposition 2.23 *Let c and d be constraints in $\mathcal{C}$: $c \vdash_{\mathcal{C}} d$ iff $c^\circ \vdash_{\mathcal{C}}{}^\circ d^\circ$.*

We now check that the translations of configurations have the expected behavior:

Proposition 2.24 *Let $(X; c; \Gamma)$ and $(Y; d; \Delta)$ be CC configurations:*
(i) $(X; c; \Gamma) \equiv (Y; d; \Delta)$ iff $(X; c^\circ; \Gamma^\circ) \equiv^\circ (Y; d^\circ; \Delta^\circ)$;
(ii) if $(X; c; \Gamma) \longrightarrow_{CC} (Y; d; \Delta)$ then $(X; c^\circ; \Gamma^\circ) \longrightarrow_{LCC} (Y; d^\circ; \Delta^\circ)$;
(iii) if $(X; c^\circ; \Gamma^\circ) \longrightarrow_{LCC} (Y; d^\circ; \Delta^\circ)$ then $(X; c; \Gamma) \longrightarrow_{CC} (Y; e; \Delta)$, with $e \vdash_{\mathcal{C}} d$.

Proof. (i) is evident.

For (ii), we proceed by induction on $\longrightarrow_{CC}$, the only interesting case is the *ask* rule: we suppose

$$(X; c; \forall \vec{y}(d \to A), \Gamma) \longrightarrow_{CC} (X; c; A[\vec{t}/\vec{y}], \Gamma),$$

using the $c \vdash_{\mathcal{C}} d[\vec{t}/\vec{y}]$ relation. We thus have $c \vdash_{\mathcal{C}} c \wedge d[\vec{t}/\vec{y}]$, and from proposition 2.23, $c^\circ \vdash_{\mathcal{C}}{}^\circ (c \wedge d[\vec{t}/\vec{y}])^\circ = c^\circ \otimes d[\vec{t}/\vec{y}]^\circ$. As a consequence

$$(X; c^\circ; \forall \vec{y}(d^\circ \to A^\circ), \Gamma^\circ) \longrightarrow_{LCC} (X; c^\circ; A^\circ[\vec{t}/\vec{y}], \Gamma^\circ),$$

qed.

For (iii), we proceed by induction on $\longrightarrow_{LCC}$. The only interesting case is again the *ask* rule: we suppose

$$(X; c^\circ; \forall\vec{y}(d^\circ \to A^\circ), \Gamma^\circ) \longrightarrow_{LCC} (X; e^\circ; A^\circ[\vec{t}/\vec{y}], \Gamma^\circ),$$

using the relation $c^\circ \vdash_{\mathcal{C}^\circ} d^\circ[\vec{t}/\vec{y}] \otimes e^\circ = (d \wedge e)^\circ$. Thus from the proposition 2.23, $c \vdash_{\mathcal{C}} d[\vec{t}/\vec{y}] \wedge e \vdash_{\mathcal{C}} d[\vec{t}/\vec{y}]$, hence

$$(X; c; \forall\vec{y}(d \to A), \Gamma) \longrightarrow_{CC} (X; c; A[\vec{t}/\vec{y}], \Gamma),$$

and $c \vdash_{\mathcal{C}} e$, qed. □

The above translation is correct w.r.t. the observation of the stores and of the successes of a CC computation (case (i) and (ii) of the proposition 2.24).

2.2.3 Example of LCC program

A classical benchmark of expressiveness for concurrent languages is the dining philosophers: `N` philosophers are sitting around a table and alternate thinking and eating. Each one of them has a fork on his right, and thus also on his left, and needs these two forks to eat (the chop-sticks version may be more realistic).

As suggested in [3], this problem has an extremely simple solution in LCC.

The atomic constraints are: `fork(I)`, `eat(I,N)` for `I,N`$\in \mathbb{N}$, and `N=M`, `N`$\neq$`M` for `N,M` $\in \mathbb{N}$. The linear constraint system in this example is thus a combination of (the translation of) standard equality constraints over $(\mathbb{N}, +)$, and of linear constraints tokens `fork` and `eat` with no other non-logical axioms than the equality axiom schema: $c(\vec{x}) \otimes !(\vec{x} = \vec{y}) \Vdash c(\vec{y})$ for any constraint predicate c.

```
philosopher(I,N) =
      fork(I) ⊗ fork(I+1 mod N) →
           (tell(eat(I,N)) ||
           eat(I,N) →
                (tell(fork(I) ⊗ fork(I+1 mod N)) ||
                philosopher(I,N))).

recphilo(M,P) =
      M ≠ P → (philosopher(M,P) || tell(fork(M)) ||
                recphilo(M+1,P)) ||
      M = P → (philosopher(M,P) || tell(fork(M))).

init(N) = recphilo(1,N).
```

An execution with initial agent `init(5)` for example will install the philosophers and the forks, and the (infinite) sequence of stores along a derivation will contain various eating periods for the philosophers according to the scheduling of agents in parallel composition.

It is worth noting that unlike in a classical CC program, the imperative data structures are encoded directly with linear constraints, instead of streams [32], and that unlike the Linda version of [4] there is no need for introducing "tickets", as the guard in the ask can be the tensor product of both forks.

This program enjoys safety and liveness properties: two adjacent philosophers cannot eat at the same time, and at least one philosopher can eat.

A proof of safety of this program is given in section 4, using the phase semantics of linear logic.

3 Logical semantics

3.1 Characterizing CC stores in intuitionistic logic

Let us fix a constraint system $(\mathcal{C}, \Vdash_{\mathcal{C}})$ and a set of declarations $\mathcal{D}$.

Definition 3.1 *Deterministic CC agents are translated into intuitionistic formulas in the following way:*

$$
\begin{array}{cc}
\multicolumn{2}{c}{tell(c)^\dagger = c} \\
p(\vec{x})^\dagger = p(\vec{x}) & (\exists x A)^\dagger = \exists x A^\dagger \\
(\forall \vec{x}(c \to A))^\dagger = \forall \vec{x}(c \Rightarrow A^\dagger) & (A \parallel B)^\dagger = A^\dagger \wedge B^\dagger
\end{array}
$$

If Γ is the multiset of agents $(A_1 \dots A_n)$, one defines $\Gamma^\dagger = A_1^\dagger \wedge \dots \wedge A_n^\dagger$. If $\Gamma = \emptyset$ then $\Gamma^\dagger = 1$.

The translation $(X; c; \Gamma)^\dagger$ of a configuration is the formula $\exists X(c \wedge \Gamma^\dagger)$.

IL($\mathcal{C}$,$\mathcal{D}$) denotes the deduction system obtained by adding to IL:

- the non-logical axiom $c \vdash d$ for every $c \Vdash_{\mathcal{C}} d$ in $\Vdash_{\mathcal{C}}$,
- the non-logical axiom $p(\vec{x}) \vdash A^\dagger$ for every declaration[3] $p(\vec{x}) = A$ in $\mathcal{D}$.

$\dashv\vdash$ denotes logical equivalence.

Theorem 3.2 (Soundness) *Let $(X; c; \Gamma)$ and $(Y; d; \Delta)$ be deterministic CC configurations.*
If $(X; c; \Gamma) \equiv (Y; d; \Delta)$ then $(X; c; \Gamma)^\dagger \dashv\vdash_{IL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\dagger$.
If $(X; c; \Gamma) \longrightarrow^*_{CC} (Y; d; \Delta)$ then $(X; c; \Gamma)^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\dagger$.

Proof. By induction on $\equiv$ and $\longrightarrow_{CC}$.

- For *parallel composition*, *α-conversion* and *$\mathcal{C}$-equivalence*, it is immediate.
- For *hiding*, $\exists x(A \wedge B) \dashv\vdash A \wedge \exists x B$ and $\exists x A \dashv\vdash A$ if $x \notin fv(A)$.
- For *tell*, *congruence* and *procedure calls*, it is immediate.
- For *ask*, just note that $c \wedge \forall \vec{x}(d \Rightarrow A) \vdash c \wedge A[\vec{t}/\vec{y}]$ if $c \Vdash_{\mathcal{C}} d[\vec{t}/\vec{y}]$, qed.

□

The converse is true for the observation of stores. Let $\kappa = (X; c; \Gamma)$ be a deterministic CC configuration, and ϕ be a constraint or a procedure call. $\kappa \stackrel{\gg}{\longrightarrow} \phi$ stands for:

[3] Translating CC declarations with a logical equivalence instead of an implication would preserve both soundness and completeness results of this section, but the intermediate lemmas need be generalized in order to take into account the foldings of procedure declarations (i.e. replacing a formula by the procedure it defines) that would become possible in the logic, although they have no operational counterpart.

- if ϕ is a constraint: "there exists a configuration $(Y; d; \Delta)$, such that $\exists Y d \vdash_{\mathcal{C}} \phi$ and $\kappa \longrightarrow^*_{CC} (Y; d; \Delta)$",
- if ϕ is a procedure call: "there exists a configuration $(Y; d; \phi, \Delta)$, such that $fv(\phi) \cap Y = \emptyset$ and $\kappa \longrightarrow^*_{CC} (Y; d; \phi, \Delta)$".

Lemma 3.3 *Let κ and λ be two deterministic CC configurations such that $\kappa^\dagger = \lambda^\dagger$, and ϕ a constraint or a procedure call.*
$\kappa \xrightarrow{\gg} \phi$ iff $\lambda \xrightarrow{\gg} \phi$.

Proof. We prove the lemma by induction on the formula $\kappa^\dagger = \lambda^\dagger$.

- If $\kappa^\dagger = \lambda^\dagger$ is atomic, it is clear.
- If $\kappa^\dagger = \lambda^\dagger = \forall \vec{x}(c \Rightarrow A^\dagger)$, with c a constraint and A an agent, then κ and λ are necessarily both equal to $(\emptyset; 1; \forall \vec{x}(c \rightarrow A))$.
- If $\kappa^\dagger = \lambda^\dagger = \exists x A^\dagger$, then the only two possibilities for κ and λ are $(\{x\} \cup Y; c; \Gamma)$ and $(\emptyset; 1; \exists x \exists Y(tell(c) \parallel \Gamma))$. One implication is thus obvious and the other one a simple corollary of the monotonicity property 2.19.
- If $\kappa^\dagger = \lambda^\dagger = A^\dagger \wedge B^\dagger$, then the four possibilities for κ and λ are: $(\emptyset; 1; \Gamma, \Delta)$ (with $\Gamma^\dagger = A^\dagger$ and $\Delta^\dagger = B^\dagger$), $(\emptyset; 1; A \parallel B)$, $(\emptyset; c; B)$ (if $A^\dagger = c$, a constraint, i.e. $A = c$ or $A = tell(c)$) and $(\emptyset; c \wedge d; \emptyset)$ (if $A^\dagger = c$ and $B^\dagger = d$, constraints). The induction is useful only in the first case, and the result is evident.

□

Lemma 3.4 *Let $\kappa = (X; c; \Gamma)$ be a deterministic CC configuration, and ϕ be a constraint or a procedure call.*
If $\kappa^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} \phi$, then $\kappa \xrightarrow{\gg} \phi$.

Proof. We prove the result for multisets of agents. We prove that if ${A_1}^\dagger, \ldots, {A_n}^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} \phi$, where the A_i's are agents and ϕ is either a constraint or a procedure call, then $(\emptyset; 1; A_1, \ldots, A_n) \xrightarrow{\gg} \phi$.

This is sufficient to conclude: indeed let $(X; c; \Gamma)$ be a deterministic CC configuration, and ϕ be a constraint or a procedure call. Note that $(X; c; \Gamma)^\dagger = \exists X(tell(c) \parallel \Gamma)^\dagger$. Therefore if $(X; c; \Gamma)^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} \phi$, then $(X; 1; tell(c), \Gamma)) \equiv (\emptyset; 1; \exists X(tell(c) \parallel \Gamma)) \xrightarrow{\gg} \phi$. But $(X; 1; tell(c), \Gamma))^\dagger = (X; c; \Gamma))^\dagger$. So by lemma 3.3 we will conclude $(X; c; \Gamma)) \xrightarrow{\gg} \phi$.

Let us proceed by induction on a sequent calculus proof π of ${A_1}^\dagger, \ldots, {A_n}^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} \phi$, where the A_i's are agents and ϕ is either a constraint or a procedure call. We shall consider, without loss of generality, that in π the left introduction of $\forall$ and of $\Rightarrow$ are always consecutive (if it is not the case, it is well-known that the rules can be permuted to obtain such a proof, see for instance [9]) we will thus group them as a single rule.

First remark that this induction is meaningful. Indeed the only cuts which cannot be eliminated in a proof bear on non-logical axioms, so that they are of one of the following forms:

$$\frac{\Gamma \vdash p \qquad \overline{p \vdash \phi}}{\Gamma \vdash \phi} \qquad\qquad \frac{\Gamma \vdash e \qquad \overline{e \vdash f}}{\Gamma \vdash f}$$

$$\frac{\overline{p \vdash \psi} \qquad \Gamma, \psi \vdash \phi}{\Gamma, p \vdash \phi} \qquad\qquad \frac{\overline{e \vdash f} \qquad \Gamma, f \vdash \phi}{\Gamma, e \vdash \phi}$$

Hence the application of the cut rule introduces sequents in which the new formula on the right is always either a constraint or a procedure call. On the other hand the formulas to the left remain sub-formulas of translations of agents or constraints or procedure calls, so they are agents. (Note that the induction hypothesis requires the result not only for constraints, but also for procedure calls.)

Each logical rule simulates a CC transition rule.

- π is an axiom: one uses the reflexivity of $\longrightarrow_{CC}^{*}$ in the case of a logical axiom, the rule *procedure calls* for an axiom $p \vdash q$; the case of an axiom $d \vdash_C e$ is trivial.
- π ends with a cut: the possible cases are the ones enumerated above. Let us consider for instance :

$$\frac{\Gamma^{\dagger} \vdash p \qquad \overline{p \vdash \phi}}{\Gamma^{\dagger} \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; \Gamma) \xrightarrow{\gg} p$, i.e. there exists a configuration $(Y; d; p, \Delta)$ such that $fv(p) \cap Y = \emptyset$ and $(\emptyset; 1; \Gamma) \longrightarrow_{CC}^{*} (Y; d; p, \Delta)$. Thus $(\emptyset; 1; \Gamma) \longrightarrow_{CC}^{*} (Y; d; \phi, \Delta)$, with $fv(\phi) \cap Y = \emptyset$ as $fv(\phi) \subset fv(p)$. If ϕ is a procedure call, it is finished. If ϕ is a constraint c, then the declaration is $p = \mathit{tell}(c)$ so $(\emptyset; 1; \Gamma) \longrightarrow_{CC}^{*} (Y; d \wedge c; \Delta)$, qed.

  The other case, when p is a constraint, is immediate. The other cases, where the axiom is the left premise, are similar.

- π ends with a left introduction of 1: note that $(\emptyset; 1; \Gamma, \mathit{tell}(1)) \longrightarrow_{CC}^{*} (\emptyset; 1; \Gamma)$. By induction hypothesis, $(\emptyset; 1; \Gamma) \xrightarrow{\gg} \phi$ thus $(\emptyset; 1; \Gamma, \mathit{tell}(1) \xrightarrow{\gg} \phi$, qed.
- π ends with a weakening:

$$\frac{\Gamma^{\dagger} \vdash \phi}{\Gamma^{\dagger}, A^{\dagger} \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; \Gamma) \xrightarrow{\gg} \phi$, thus $(\emptyset; 1; A, \Gamma) \xrightarrow{\gg} \phi$ thanks to the monotonicity of $\longrightarrow_{CC}$ (proposition 2.11).

- π ends with a contraction:

$$\frac{\Gamma^{\dagger}, A^{\dagger}, A^{\dagger} \vdash \phi}{\Gamma^{\dagger}, A^{\dagger} \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; A, A, \Gamma) \xrightarrow{\gg} \phi$. In this sequence of transitions, some steps activate some occurrence of A, some others activate a sub-agent of Γ. The important point is that for the deterministic agent A, the next transition in which it can further be active (in other words the next action that it can perform) is determined (no $+$). One can thus assume that in the above execution, each time one activates a sub-agent of an occurrence of A, the same transition with the other occurrence is performed just the next moment, it is of course possible thanks to proposition 2.9. Now starting from the configuration $(\emptyset; 1; A, \Gamma)$, one simulates the above execution by applying the same transitions to sub-agents of Γ and "contracting" the pairs of transitions for the sub-agents of A: one applies the rule to one of the two copies, always the same one. One then obtains an execution $(\emptyset; 1; A, \Gamma) \stackrel{\gg}{\longrightarrow} \phi$.

- π ends with:

$$\frac{\Gamma^\dagger, A^\dagger, B^\dagger \vdash \phi}{\Gamma^\dagger, A^\dagger \wedge B^\dagger \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; A, B, \Gamma) \stackrel{\gg}{\longrightarrow} \phi$. If $A^\dagger$ and $B^\dagger$ are constraints, there is a priori an ambiguity regarding the agent C whose translation is $A^\dagger \wedge B^\dagger$: C can be $tell(A \wedge B)$ or $tell(A) \parallel tell(B))$. However, as the two configurations have the same translation as $(\emptyset; 1; A, B, \Gamma)$, according to lemma 3.3, $(\emptyset; 1; C, \Gamma) \stackrel{\gg}{\longrightarrow} \phi$.

- π ends with:

$$\frac{\Gamma^\dagger \vdash \phi \qquad \Delta^\dagger \vdash \psi}{\Gamma^\dagger, \Delta^\dagger \vdash \phi \wedge \psi}$$

  $\phi \wedge \psi$ is not atomic, so ϕ and ψ are constraints. The result is now immediate: starting from the configuration $(\emptyset; 1; \Gamma, \Delta)$, one just joins the two executions $(\emptyset; 1; \Gamma) \stackrel{\gg}{\longrightarrow} \phi$ and $(\emptyset; 1; \Delta) \stackrel{\gg}{\longrightarrow} \psi$ end to end.

- π ends with a right introduction of $\exists$ (in case ϕ is a constraint): immediate.

- π ends with:

$$\frac{\Gamma^\dagger, A^\dagger \vdash \phi}{\Gamma^\dagger, \exists x A^\dagger \vdash \phi}\; x \notin fv(\Gamma, \phi)$$

  By induction hypothesis, $(\emptyset; 1; A, \Gamma) \stackrel{\gg}{\longrightarrow} \phi$. As $x \notin fv(\Gamma)$, $(\emptyset; 1; \exists x A, \Gamma) \equiv (x; 1; A, \Gamma)$, and moreover $x \notin fv(\phi)$, so by lemma 3.3, $(\emptyset; 1; \exists x A, \Gamma) \stackrel{\gg}{\longrightarrow} \phi$, qed.

- π ends with (thanks to the preliminary remark on the permutability of rules):

$$\frac{\dfrac{\Gamma^\dagger, A^\dagger[\vec{t}/\vec{x}] \vdash \phi \qquad \Delta^\dagger \vdash c[\vec{t}/\vec{x}]}{\Gamma^\dagger, \Delta^\dagger, c[\vec{t}/\vec{x}] \Rightarrow A^\dagger[\vec{t}/\vec{x}] \vdash \phi}}{\Gamma^\dagger, \Delta^\dagger, \forall \vec{x}(c \Rightarrow A^\dagger) \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; \Delta) \stackrel{\gg}{\longrightarrow} c[\vec{t}/\vec{x}]$, i.e. there exists a configuration $(Y; d; \Sigma)$, such that $d \vdash_{\mathcal{C}} c[\vec{t}/\vec{x}]$ and $(\emptyset; 1; \Delta) \longrightarrow^*_{CC} (Y; d; \Sigma)$. Thus $(\emptyset; 1; \forall \vec{x}(c \rightarrow A), \Delta) \longrightarrow^*_{CC} (Y; d; \forall \vec{x}(c \rightarrow A), \Sigma) \longrightarrow^*_{CC} (Y; d; A[\vec{t}/\vec{x}], \Sigma)$. Therefore, $(\emptyset; 1; \forall \vec{x}(c \rightarrow A), \Delta, \Gamma) \longrightarrow^*_{CC} (Y; d; A[\vec{t}/\vec{x}], \Sigma, \Gamma)$. Moreover by induction hypothesis, $(\emptyset; 1; A[\vec{t}/\vec{x}], \Gamma) \stackrel{\gg}{\longrightarrow} \phi$, whence $(\emptyset; 1; \forall \vec{x}(c \rightarrow A), \Delta, \Gamma) \stackrel{\gg}{\longrightarrow} \phi$.

□

Now, for a set S of constraints, let us note $\downarrow S = \{c \in \mathcal{C} \mid \exists d \in S, d \vdash_{\mathcal{C}} c\}$, we get:

Theorem 3.5 (Observation of deterministic stores) *Let A be a deterministic CC agent, define* $\mathcal{L}^{store}(\mathcal{C},\mathcal{D}.A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} c\}$ *we have:*

$$\mathcal{L}^{store}(\mathcal{C},\mathcal{D}.A) = \downarrow \mathcal{O}^{store}_{CC}(\mathcal{C},\mathcal{D}.A)$$

Proof. One inclusion is obvious by applying the previous theorem, it is just the definition of an accessible store, the other is a direct consequence of theorem 3.2. $\square$

The characterization of stores for non-deterministic configurations is not obvious in the framework of intuitionistic logic: indeed on one hand the simple idea of translating the choice operator $+$ by disjunction $\vee$ requires to modify the operational semantics of $+$ (e.g. for soundness, because $A \vee B \not\vdash A$), and on the other hand the idea of translating $+$ by conjunction $\wedge$ preserves the soundness, but not the characterization of stores because, for instance, the store $c \wedge d$ is not accessible from the configuration $(\emptyset; 1; tell(c) + tell(d))$.

It is possible for the observation of some *must* properties, i.e. properties that are true in all possible non-deterministic configurations, like the stores entailed in all branches of the computation, as shown here in theorem 3.15. However we shall see in the next section in the more general framework of linear CC that a logical characterization of both *must* and *may* properties is possible in linear logic w.r.t. both successes and accessible stores.

3.2 Characterizing CC and LCC stores and successes in intuitionistic linear logic

The observation of stores is important, however it represents only one aspect of the operational behavior of CC programs.

Consider the following three programs :

$$p(x) = x \geq 1$$

$$p(x) = x \geq 1 \parallel p(x)$$

$$p(x) = x \geq 1 \parallel (false \to A).$$

They define the same stores ($x \geq 1$), thus they are equivalent w.r.t. the observation of stores, whereas the first one terminates on a success, the second one loops and the third one suspends.

As is shown by the following counter-examples, neither the successes nor the suspensions are characterizable in intuitionistic logic:

- $\dashv$: In general it is false that $A \dashv B$ (where B is a success store or a suspension) implies $(\emptyset; 1; A) \longrightarrow_{LCC} (\emptyset; 1; B)$. For instance $c \to d \dashv d$ but $c \to d$ suspends in the empty store, and thus does not reduce to the success d. In the case where B is a suspension, for instance $d \parallel (c \to d)$ with d not implying c, note that $d \dashv d \wedge (c \to d)$ but $tell(d)$ doesn't reduce to that suspension.

- $\vdash$: One has similar problems with $\vdash$. We have $d \wedge (c \Rightarrow A) \vdash d$ whereas $d \parallel (c \to A)$ suspends as soon as $d \not\vdash c$. Besides $d \wedge (d \Rightarrow e) \vdash d \Rightarrow e$, but $d \parallel (d \to e)$ has a success $(d \wedge e)$ and does not suspend.

- $\dashv\vdash$: Similarly, for the equivalence $\dashv\vdash$, let us suppose that d does not imply c, and let us consider the following equivalence: $d \wedge (c \Rightarrow d) \dashv\vdash d$. One cannot conclude anything about the operational behavior of the agents $tell(d)$ and $d \parallel (c \rightarrow d)$.

The obstacle is the structural rule of (left) weakening:

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

Girard's linear logic [10] is a refinement of the contraction and weakening rules of usual logic. It seems therefore natural to interpret CC programs in linear logic.

Let us fix a linear constraint system $(\mathcal{C}, \Vdash_{\mathcal{C}})$ and a set of declarations $\mathcal{D}$.

Definition 3.6 *LCC agents are translated into linear logic formulas in the following way:*

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

If Γ is the multiset of agents $(A_1 \dots A_n)$, define $\Gamma^{\ddagger} = A_1^{\ddagger} \otimes \dots \otimes A_n^{\ddagger}$. If $\Gamma = \emptyset$ then $\Gamma^{\ddagger} = 1$. The translation $(X; c; \Gamma)^{\ddagger}$ of a configuration is the formula $\exists X (c \otimes \Gamma^{\ddagger})$.

ILL($\mathcal{C}$,$\mathcal{D}$) denotes the deduction system obtained by adding to ILL:

- the non-logical axiom $c \vdash d$ for every $c \Vdash_{\mathcal{C}} d$ in $\Vdash_{\mathcal{C}}$,
- the non-logical axiom $p(\vec{x}) \vdash A^{\ddagger}$ for every declaration[4] $p(\vec{x}) = A$ in $\mathcal{D}$.

Theorem 3.7 (Soundness) *Let $(X; c; \Gamma)$ and $(Y; d; \Delta)$ be LCC configurations.*
If $(X; c; \Gamma) \equiv (Y; d; \Delta)$ then $(X; c; \Gamma)^{\ddagger} \dashv\vdash_{ILL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^{\ddagger}$.
If $(X; c; \Gamma) \longrightarrow^{*}_{LCC} (Y; d; \Delta)$ then $(X; c; \Gamma)^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^{\ddagger}$.

Proof. The proof is essentially the same as in intuitionistic logic. Note that for the operator of choice $+$ translated by the additive conjunction $\&$, $A \& B \vdash A$ and $A \& B \vdash B$. □

Conversely, one can characterize the observation of successes, even in presence of the operator of explicit choice $+$:

Notation:
Let $\kappa = (X; c; \Gamma)$ be an LCC configuration, and ϕ be a constraint or a procedure call. $\kappa \xrightarrow{\ggg} \phi$ stands for:

- if ϕ is a constraint: "there exists a configuration $(Y; d; \emptyset)$, such that $\exists Y d \vdash_{\mathcal{C}} \phi$ and $\kappa \longrightarrow^{*}_{LCC} (Y; d; \emptyset)$",
- if ϕ is a procedure call: "there exists a configuration $(Y; d; \phi)$, such that $fv(\phi) \cap Y = \emptyset$, $\exists Y d \vdash_{\mathcal{C}} 1$ and $\kappa \longrightarrow^{*}_{LCC} (Y; d; \phi)$".

[4] Here again the CC declarations could be translated with an equivalence instead of an implication without affecting the main soundness and completeness theorems of this section.

Lemma 3.8 *Let κ and λ be two configurations LCC such that $\kappa^\ddagger = \lambda^\ddagger$, and ϕ be a constraint or a procedure call.*

$\kappa \xrightarrow{\ggg} \phi$ *iff* $\lambda \xrightarrow{\ggg} \phi$.

Proof. Again the proof is essentially the same as in intuitionistic logic, the difference between κ and λ amounts to non performed *tell*'s, or to $tell(c \otimes d)$ vs $tell(c) \parallel tell(d)$. □

Lemma 3.9 *Let $(\emptyset; 1; \Gamma)$ be an LCC configuration and ϕ be a constraint or a procedure call such that $(\emptyset; 1; \Gamma) \xrightarrow{\ggg} \phi$. Let x be a variable such that $x \notin fv(\phi)$, then $(x; 1; \Gamma) \xrightarrow{\ggg} \phi$*

Proof. Let Δ be the agents obtained by replacing all existentially quantified occurrences of x in Γ by a fresh variable z. We have $(\emptyset; 1; \Gamma) \equiv (\emptyset; 1; \Delta)$ thus $(\emptyset; 1; \Delta) \xrightarrow{\ggg} \phi$.

If ϕ is a constraint then $(\emptyset; 1; \Delta) \longrightarrow^* (Y; d; \emptyset)$ with $\exists Y d \vdash_{\mathcal{C}} \phi$. One can then show by an easy induction on this derivation that $(\{x\}; 1; \Delta) \longrightarrow^* (\{x\} \cup Y; d; \emptyset)$ (the only non-trivial case is the use of the Hiding rule, but as x has no bounded occurrence in Δ, the same rule can be used in the derivation for $(\{x\} \cup Y; d; \emptyset)$). As $x \notin fv(\phi)$ we get $\exists \{x\} \cup Y\ d \vdash_{\mathcal{C}} \phi$ so $(\{x\}; 1; \Delta) \xrightarrow{\ggg} \phi$. If ϕ is a procedure call we also get $(\{x\}; 1; \Delta) \xrightarrow{\ggg} \phi$ in the same manner. The result then follows from the observation that $(\{x\}; 1; \Delta) \equiv (\{x\}; 1; \Gamma)$. □

Lemma 3.10 *Let $\kappa = (X; c; \Gamma)$ be an LCC configuration, and ϕ be a constraint or a procedure call.*

If $\kappa^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \phi$, then $\kappa \xrightarrow{\ggg} \phi$.

Proof. The result is proved for multisets of agents:
if ${A_1}^\ddagger, \ldots, {A_n}^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \phi$, then $(\emptyset; 1; A_1, \ldots, A_n) \xrightarrow{\ggg} \phi$.

This is sufficient to conclude: indeed let $(X; c; \Gamma)$ be a configuration, and ϕ be a constraint or a procedure call. Note that $(X; c; \Gamma)^\ddagger = \exists X(tell(c) \parallel \Gamma)^\ddagger$. Therefore if $(X; c; \Gamma)^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \phi$, then $(X; 1; tell(c), \Gamma)) \equiv (\emptyset; 1; \exists X(tell(c) \parallel \Gamma)) \xrightarrow{\ggg} \phi$. But $(X; 1; tell(c), \Gamma))^\ddagger = (X; c; \Gamma))^\ddagger$. So according to lemma 3.8, $(X; c; \Gamma) \xrightarrow{\ggg} \phi$.

The proof proceeds by induction on a sequent calculus proof π of $A_1^\ddagger, \ldots, A_n^\ddagger \vdash \phi$. 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 instance [20], noting that !, that is the only case of unpermutability with $\forall$, appears only in the constraint part and thus not below a $\multimap$), we will thus group them as a single rule, so that each logical rule simulates an LCC transition rule.

- π is an axiom: one uses the reflexivity of $\longrightarrow^*$ in the case of a logical axiom, the rule for procedure calls for an axiom $p \vdash q$; the case of an axiom $d \vdash_{\mathcal{C}} e$ is trivial.
- π ends with a cut. The only cuts which cannot be eliminated in a proof bear on non-logical axioms, let us consider for instance a cut with procedure declaration:

$$\frac{\Gamma^\ddagger \vdash p \quad \overline{p \vdash \phi}}{\Gamma^\ddagger \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; \Gamma) \xrightarrow{\ggg} p$, i.e. there exists a configuration $(Y; d; p)$ such that $fv(p) \cap Y = \emptyset$, $\exists Y d \vdash_{\mathcal{C}} 1$ and $(\emptyset; 1; \Gamma) \longrightarrow^* (Y; d; p)$. Thus $(\emptyset; 1; \Gamma) \longrightarrow^* (Y; d; \phi)$;

$fv(\phi) \cap Y = \emptyset$ as $fv(\phi) \subset fv(p)$. If ϕ is a procedure call, it is finished. If ϕ is a constraint c, then $(\emptyset; 1; \Gamma) \longrightarrow^* (Y; d \otimes c; \emptyset)$, $fv(c) \cap Y = \emptyset$ and $\exists Y d \otimes c \vdash_{\mathcal{C}} c$, qed.

The other case, when p is a constraint, is immediate. The other cases, where the axiom is the left premise, are similar.

- π ends with a left introduction of 1: note that $(\emptyset; 1; \Gamma, tell(1)) \longrightarrow^* (\emptyset; 1; \Gamma)$. By induction hypothesis, $(\emptyset; 1; \Gamma) \xrightarrow{\ggg} \phi$ thus $(\emptyset; 1; \Gamma, tell(1)) \xrightarrow{\ggg} \phi$, qed.

- π ends with:

$$\frac{\Gamma^\ddagger, A^\ddagger, B^\ddagger \vdash \phi}{\Gamma^\ddagger, A^\ddagger \otimes B^\ddagger \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; A, B, \Gamma) \xrightarrow{\ggg} \phi$. If $A^\ddagger$ and $B^\ddagger$ are constraints, there is a priori an ambiguity on the agent C whose translation is $A^\ddagger \otimes B^\ddagger$: it can be $tell(A \otimes B)$ or $tell(A) \parallel tell(B))$. However, as the two configurations have the same translation as $(\emptyset; 1; A, B, \Gamma)$, according to lemma 3.8, $(\emptyset; 1; C, \Gamma) \xrightarrow{\ggg} \phi$.

- π ends with:

$$\frac{\Gamma^\ddagger \vdash \phi \quad \Delta^\ddagger \vdash \psi}{\Gamma^\ddagger, \Delta^\ddagger \vdash \phi \otimes \psi}$$

  $\phi \otimes \psi$ is not atomic, so ϕ and ψ are constraints. The result is now immediate: starting from the configuration $(\emptyset; 1; \Gamma, \Delta)$, one just joins the two executions $(\emptyset; 1; \Gamma) \xrightarrow{\ggg} \phi$ and $(\emptyset; 1; \Delta) \xrightarrow{\ggg} \psi$ end to end.

- π ends with:

$$\frac{\Gamma^\ddagger, A^\ddagger \vdash \phi}{\Gamma^\ddagger, A^\ddagger \,\&\, B^\ddagger \vdash \phi}$$

  By induction hypothesis, $(\emptyset; 1; A, \Gamma) \xrightarrow{\ggg} \phi$, Now $(\emptyset; 1; A + B, \Gamma) \longrightarrow^* (\emptyset; 1; A, \Gamma)$, therefore $(\emptyset; 1; A + B, \Gamma) \xrightarrow{\ggg} \phi$.

- π ends with a right introduction of $\exists$ (in case ϕ is a constraint): immediate.

- π ends with:

$$\frac{\Gamma^\ddagger, A^\ddagger \vdash \phi}{\Gamma^\ddagger, \exists x A^\ddagger \vdash \phi}\; x \notin fv(\Gamma, \phi)$$

  By induction hypothesis, $(\emptyset; 1; A, \Gamma) \xrightarrow{\ggg} \phi$. As $x \notin fv(\Gamma)$, $(\emptyset; 1; \exists x A, \Gamma) \longrightarrow (x; 1; A, \Gamma)$, and moreover $x \notin fv(\phi)$, so by lemma 3.9, $(\emptyset; 1; \exists x A, \Gamma) \xrightarrow{\ggg} \phi$, qed.

- π ends with (thanks to the preliminary remark on the permutability of rules):

$$\frac{\dfrac{\Gamma^\ddagger, A^\ddagger[\vec{t}/\vec{x}] \vdash \phi \quad \Delta^\ddagger \vdash c[\vec{t}/\vec{x}]}{\Gamma^\ddagger, \Delta^\ddagger, (c[\vec{t}/\vec{x}] \multimap A^\ddagger[\vec{t}/\vec{x}]) \vdash \phi}}{\Gamma^\ddagger, \Delta^\ddagger, \forall \vec{x}(c \multimap A^\ddagger) \vdash \phi}$$

By induction hypothesis, $(\emptyset;1;\Delta) \overset{\ggg}{\longrightarrow} c[\vec{t}/\vec{x}]$, i.e. there exists a configuration $(Y;d;\emptyset)$, such that $\exists Y d \vdash_{\mathcal{C}} c[\vec{t}/\vec{x}]$ and $(\emptyset;1;\Delta) \longrightarrow^* (Y;d;\emptyset)$. Thus by applying the *ask* rule, one obtains $(\emptyset;1;\forall\vec{x}(c \to A),\Delta) \longrightarrow^* (Y;d;\forall\vec{x}(c \to A)) \longrightarrow^* (Y;1;A[\vec{t}/\vec{x}])$. Therefore, $(\emptyset;1;\forall\vec{x}(c \to A),\Delta,\Gamma) \longrightarrow^* (Y;1;A[\vec{t}/\vec{x}],\Gamma)$. Moreover by induction hypothesis, $(\emptyset;1;A[\vec{t}/\vec{x}],\Gamma) \overset{\ggg}{\longrightarrow} \phi$, whence $(\emptyset;1;\forall\vec{x}(c \to A),\Delta,\Gamma) \overset{\ggg}{\longrightarrow} \phi$.

- π ends with a dereliction:

$$\frac{\Gamma^{\ddagger}, c \vdash \phi}{\Gamma^{\ddagger}, !c \vdash \phi}$$

  It is clear, just recall that $!c \vdash c$.

- π ends with a promotion: in that case all the formulas are necessarily constraints, therefore it is immediate.

- π ends with a weakening:

$$\frac{\Gamma^{\ddagger} \vdash \phi}{\Gamma^{\ddagger}, !c \vdash \phi}$$

  with c a constraint. By induction hypothesis, $(\emptyset;1;\Gamma) \overset{\ggg}{\longrightarrow} \phi$, so $(\emptyset;1;\mathit{tell}(!c),\Gamma) \overset{\ggg}{\longrightarrow} \phi$ (one performs the *tell*, noting that $!c \vdash_{\mathcal{C}} 1$).

- π ends with a contraction:

$$\frac{\Gamma^{\ddagger}, !c, !c \vdash \phi}{\Gamma^{\ddagger}, !c \vdash \phi}$$

  with c a constraint. By induction hypothesis, $(\emptyset;1;\mathit{tell}(!c),\mathit{tell}(!c),\Gamma) \overset{\ggg}{\longrightarrow} \phi$. Obviously having two occurrences of the agent $\mathit{tell}(!c)$ changes nothing because $!c \otimes !c \dashv\vdash !c$. Therefore $(\emptyset;1;\mathit{tell}(!c),\Gamma) \overset{\ggg}{\longrightarrow} \phi$ holds as well.

□

Theorem 3.11 (Observation of successes) *Let A be an LCC agent and c be a linear constraint. Define $\mathcal{LL}^{success}(\mathcal{C},\mathcal{D}.A) = \{c \in \mathcal{C} | A^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c\}$. We have*

$$\mathcal{LL}^{success}(\mathcal{C},\mathcal{D}.A) = \downarrow \mathcal{O}^{success}_{LCC}(\mathcal{C},\mathcal{D}.A).$$

Proof. Evident by applying the previous lemma to the configuration $(\emptyset;1;A)$. □

Let, for a set S of linear constraints, $\Downarrow S = \{c \in \mathcal{C} \mid \exists d \in S, d > c\}$.

Theorem 3.12 (Observation of stores) *Let A be an LCC agent and c be a linear constraint. Define $\mathcal{LL}^{store}(\mathcal{C},\mathcal{D}.A) = \{c \in \mathcal{C} | A^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top\}$. We have*

$$\mathcal{LL}^{store}(\mathcal{C},\mathcal{D}.A) = \Downarrow \mathcal{O}^{store}_{LCC}(\mathcal{C},\mathcal{D}.A).$$

Proof. Simply use theorem 3.11, above the right introduction of the tensor connective in $c \otimes \top$ and note that the property is preserved by left introduction rules. □

Thanks to the translation of CC into LCC (proposition 2.24), this characterization of stores and successes in linear logic holds also for CC.

3.3 Must properties

So far we have been concerned with "May" properties of LCC programs, i.e. properties that stand for some branch of the derivation tree. "Must" properties, i.e. properties that are true on all branches of the derivation tree, are also to be considered, for instance when looking at liveness properties. We show that "must" stores and "must" successes can be characterized logically using disjunction. The operational semantics has to be adapted to multisets of configurations, called frontiers, which keep track of all the alternatives in the derivation tree. For the sake of simplicity however, we will not handle the possible source of non-determinism due to the (linear) ask operations with the universal quantifier. The "must" properties modelized in this section are thus relative to the choice operator not to the indeterminism coming from the linear ask (which can be made deterministic as remarked above if only classical and synchronization constraints are used in guards).

3.3.1 Frontier calculus

A *frontier* Φ is a multiset of configurations, noted $\langle \kappa_1, \cdots, \kappa_n \rangle$ where each κ_i is a configuration $(X_i; c_i; \Gamma_i)$. The structural congruence $\equiv$ on configurations is kept unchanged. The transition relation $\longrightarrow_{LCC}$ is extended to a transition relation $\Longrightarrow_{LCC}$ on frontiers in the obvious way, the only difference being for the non-deterministic choice. The frontier transition relation $\Longrightarrow_{LCC}$ is defined as the least relation satisfying the rules in table 3.

LinearTell	$\langle (X; c; tell(d), \Gamma), \Phi \rangle \Longrightarrow_{LCC} \langle (X; c \otimes d; \Gamma), \Phi \rangle$
LinearAsk	$\dfrac{c \vdash_{\mathcal{C}} d \otimes e[\vec{t}/\vec{y}]}{\langle (X; c; \forall \vec{y}(e \rightarrow A), \Gamma), \Phi \rangle \Longrightarrow_{LCC} \langle (X; d; A[\vec{t}/\vec{y}], \Gamma), \Phi \rangle}$
Procedure calls	$\dfrac{(p(\vec{y}) = A) \in P}{\langle (X; c; p(\vec{y}), \Gamma), \Phi \rangle \Longrightarrow_{LCC} \langle (X; c; A, \Gamma), \Phi \rangle}$
$\equiv$	$\dfrac{(X; c; \Gamma) \equiv (X'; c'; \Gamma') \Longrightarrow_{LCC} (Y'; d'; \Delta') \equiv (Y; d; \Delta)}{\langle (X; c; \Gamma), \Phi \rangle \Longrightarrow_{LCC} \langle (Y; d; \Delta), \Phi \rangle}$
Blind choice	$\langle (X; c; A + B, \Gamma), \Phi \rangle \Longrightarrow_{LCC} \langle (X; c; A, \Gamma), (X; c; B, \Gamma), \Phi \rangle$

Table 3: LCC frontier transition relation

The frontier transition relation $\Longrightarrow_{CC}$ is defined similarly with *Tell* and *Ask* rules instead of the *LinearTell* and *LinearAsk* rules. Now the operational semantics for must properties is defined as follows:

Definition 3.13 (Frontier operational semantics)

$$\begin{aligned}
\mathcal{O}^{store}_{CC''}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} \mid\ & (\emptyset; 1; A) \Longrightarrow^*_{CC} \langle (X_1; d_1; \Gamma_1), \cdots, (X_n; d_n; \Gamma_n) \rangle, \\
& \exists X_1 d_1 \vdash_{\mathcal{C}} c, \ldots, \exists X_n d_n \vdash_{\mathcal{C}} c\} \\
\mathcal{O}^{store}_{LCC''}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} \mid\ & (\emptyset; 1; A) \Longrightarrow^*_{LCC} \langle (X_1; d_1; \Gamma_1), \cdots, (X_n; d_n; \Gamma_n) \rangle, \\
& \exists X_1 d_1 \vdash_{\mathcal{C}} c, \ldots, \exists X_n d_n \vdash_{\mathcal{C}} c\} \\
\mathcal{O}^{success}_{LCC''}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} \mid\ & (\emptyset; 1; A) \Longrightarrow^*_{LCC} \langle (X_1; d_1; \emptyset), \cdots, (X_n; d_n; \emptyset) \rangle, \\
& \exists X_1 d_1 \vdash_{\mathcal{C}} c, \ldots, \exists X_n d_n \vdash_{\mathcal{C}} c\}
\end{aligned}$$

3.3.2 Logical semantics.

The translation † (resp. ‡) of CC (resp. LCC) configurations into intuitionistic (resp. linear) formulas is changed for disjunctive agents and extended to frontiers in the obvious way:

$$
\begin{array}{ll}
\multicolumn{2}{c}{\langle \kappa_1, \cdots, \kappa_n \rangle^{\dagger\dagger} = (\kappa_1)^{\dagger\dagger} \vee \cdots \vee (\kappa_n)^{\dagger\dagger}} \\
(A+B)^{\dagger\dagger} = A^{\dagger\dagger} \vee B^{\dagger\dagger} & tell(c)^{\dagger\dagger} = c \\
p(\vec{x})^{\dagger\dagger} = p(\vec{x}) & (\exists x A)^{\dagger\dagger} = \exists x A^{\dagger\dagger} \\
(\forall \vec{x}(c \rightarrow A))^{\dagger\dagger} = \forall \vec{x}(c \Rightarrow A^{\dagger\dagger}) & (A \parallel B)^{\dagger\dagger} = A^{\dagger\dagger} \wedge B^{\dagger\dagger}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{c}{\langle \kappa_1, \cdots, \kappa_n \rangle^{\ddagger\ddagger} = (\kappa_1)^{\ddagger\ddagger} \oplus \cdots \oplus (\kappa_n)^{\ddagger\ddagger}} \\
(A+B)^{\ddagger\ddagger} = A^{\ddagger\ddagger} \oplus B^{\ddagger\ddagger} & tell(c)^{\ddagger\ddagger} = c \\
p(\vec{x})^{\ddagger\ddagger} = p(\vec{x}) & (\exists x A)^{\ddagger\ddagger} = \exists x A^{\ddagger\ddagger} \\
(\forall \vec{x}(c \rightarrow A))^{\ddagger\ddagger} = \forall \vec{x}(c \multimap A^{\ddagger\ddagger}) & (A \parallel B)^{\ddagger\ddagger} = A^{\ddagger\ddagger} \otimes B^{\ddagger\ddagger}
\end{array}
$$

The soundness of the translation is proved by a simple induction:

Theorem 3.14 (Soundness of frontier transitions) *Let Φ and Ψ be two frontiers. If $\Phi \Longrightarrow_{CC} \Psi$ then $\Phi^{\dagger\dagger} \vdash_{IL(\mathcal{C},\mathcal{D})} \Psi^{\dagger\dagger}$. If $\Phi \Longrightarrow_{LCC} \Psi$ then $\Phi^{\ddagger\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} \Psi^{\ddagger\ddagger}$.*

The converse doesn't hold in general because some logical deductions have no operational counterpart, like for example weakening in IL, or $A \otimes (c \multimap B) \vdash_{ILL} c \multimap (A \otimes B)$ in ILL, see [31]. Nevertheless completeness holds for the observation of CC frontier stores in IL, and for the observation of both successes and frontier stores in ILL.

Theorem 3.15 (Observation of frontier stores in IL) *Let A be a CC agent, define $\mathcal{L}''^{store}(\mathcal{C},\mathcal{D}.A) = \{c \in \mathcal{C} \mid A^{\dagger\dagger} \vdash_{IL(\mathcal{C},\mathcal{D})} c\}$ we have:*

$$\mathcal{L}''^{store}(\mathcal{C},\mathcal{D}.A) =\downarrow \mathcal{O}^{store}_{CC''}(\mathcal{C},\mathcal{D}.A)$$

Proof. One inclusion is shown by induction on a proof of $A^{\dagger} \vdash_{IL(\mathcal{C},\mathcal{D})} c$ as in lemma 3.4. Compared to lemma 3.4, there is just an additional induction step:

$$\frac{\Gamma^{\dagger\dagger}, A^{\dagger\dagger} \vdash \phi \qquad \Gamma^{\dagger\dagger}, B^{\dagger\dagger} \vdash \phi}{\Gamma^{\dagger\dagger}, A^{\dagger\dagger} \vee B^{\dagger\dagger} \vdash \phi}$$

By induction hypothesis, $(\emptyset; 1; A, \Gamma) \stackrel{\gg}{\Longrightarrow} \phi$ and $(\emptyset; 1; B, \Gamma) \stackrel{\gg}{\Longrightarrow} \phi$. Now $(\emptyset; 1; A+B, \Gamma) \Longrightarrow^* \langle(\emptyset; 1; A, \Gamma), (\emptyset; 1; B, \Gamma)\rangle$, and therefore $(\emptyset; 1; A+B, \Gamma) \stackrel{\gg}{\Longrightarrow} \phi$, qed.

As usual, the other inclusion is a direct consequence of the soundness theorem. □

Theorem 3.16 (Observation of frontier stores and successes in ILL) *Let A be an LCC agent define $\mathcal{LL}''^{store}(\mathcal{C},\mathcal{D}.A) = \{c \in \mathcal{C} \mid A^{\ddagger\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top\}$ and $\mathcal{LL}''^{success}(\mathcal{C},\mathcal{D}.A) = \{c \in \mathcal{C} \mid A^{\ddagger\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c\}$ we have:*

$$\mathcal{LL}''^{store}(\mathcal{C},\mathcal{D}.A) =\Downarrow \mathcal{O}^{store}_{LCC''}(\mathcal{C},\mathcal{D}.A) \qquad \mathcal{LL}''^{success}(\mathcal{C},\mathcal{D}.A) =\downarrow \mathcal{O}^{success}_{LCC''}(\mathcal{C},\mathcal{D}.A)$$

Proof. For the first inclusion, we first prove the result for successes, for stores we apply the same argument as in the proof of theorem 3.12. Proceed by induction on a proof of $A^{\ddagger\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c$ as in lemma 3.10. The only difference with the proof of lemma 3.10 is the following induction step:

$$\frac{\Gamma^{\ddagger\ddagger}, A^{\ddagger\ddagger} \vdash \phi \qquad \Gamma^{\ddagger\ddagger}, B^{\ddagger\ddagger} \vdash \phi}{\Gamma^{\ddagger\ddagger}, A^{\ddagger\ddagger} \oplus B^{\ddagger\ddagger} \vdash \phi}$$

By induction hypothesis, $(\emptyset; 1; A, \Gamma) \stackrel{\ggg}{\Longrightarrow} \phi$ and $(\emptyset; 1; B, \Gamma) \stackrel{\ggg}{\Longrightarrow} \phi$. Now $(\emptyset; 1; A + B, \Gamma) \Longrightarrow^* (\emptyset; 1; A, \Gamma) + (\emptyset; 1; B, \Gamma)$, and therefore $(\emptyset; 1; A + B, \Gamma) \stackrel{\ggg}{\Longrightarrow} \phi$, qed. The second inclusion is obtained via soundness. □

4 Phase semantics

4.1 Phase semantics of intuitionistic linear logic

Phase semantics is the natural provability semantics of linear logic [10]. It will also be useful for proving safety properties of CC programs, through the links between linear logic and CC. We only need here a fragment of intuitionistic linear logic ($\otimes$, $\&$ and $\multimap$, which correspond respectively to the parallel, choice and blocking ask operators, as shown in section 3.2). Nevertheless it is simpler to recall Okada's definition of the phase semantics for full intuitionistic LL [25] and to extend it to constants ($\mathbf{1}$, $\mathbf{0}$, $\top$).

The definition of formulas and the sequent calculus are recalled in appendix A.

Definition 4.1 *A* phase space $\mathbf{P} = (P, \cdot, 1, \mathcal{F})$ *is a commutative monoid* $(P, \cdot, 1)$ *together with a set* $\mathcal{F}$ *of subsets of* P*, whose elements are called* facts*, satisfying the following closure properties:*

- *$\mathcal{F}$ is closed under arbitrary intersection,*
- *for all $A \subset P$, for all $F \in \mathcal{F}$, the set $\{x \in P : \forall a \in A, a \cdot x \in F\}$ is a fact of $\mathcal{F}$, noted $A \multimap F$.*

As we shall see, facts correspond to ILL formulas and thus to LCC agents (cf. section 3.2).

Note that facts are closed under linear implication $\multimap$. Here are a few noticeable facts: the greatest fact $\top = P$, the smallest fact $\mathbf{0}$, and $\mathbf{1} = \bigcap\{F \in \mathcal{F} : 1 \in F\}$.

A *parametric fact* A is a total function from V to $\mathcal{F}$ assigning to each variable x a fact $A(x)$. Any fact can be seen as a constant parametric fact, and any operation defined on facts can be extended to parametric facts: $(A \star B)(x) = A(x) \star B(x)$.

Let A, B be (parametric) facts, define the following facts:

$$\begin{aligned}
A \,\&\, B &= A \cap B, \\
A \otimes B &= \bigcap\{F \in \mathcal{F} : A \cdot B \subset F\}, \\
A \oplus B &= \bigcap\{F \in \mathcal{F} : A \cup B \subset F\}, \\
\exists x A &= \bigcap\{F \in \mathcal{F} : (\bigcup_{x \in V} A(x)) \subset F\}, \\
\forall x A &= \bigcap\{F \in \mathcal{F} : (\bigcap_{x \in V} A(x)) \subset F\}.
\end{aligned}$$

Definition 4.2 *An* enriched phase space *is a phase space* $(P, \cdot, 1, \mathcal{F})$ *together with a subset* O *of* $\mathcal{F}$*, whose elements are called* open facts*, such that:*

- *O is closed under arbitrary $\oplus$ (in particular there is a greatest open fact),*
- *$\mathbf{1}$ is the greatest open fact,*
- *O is closed under finite $\otimes$,*
- *$\otimes$ is idempotent on O (if $A \in O$ then $A \otimes A = A$).*

$!A$ is defined as the greatest open fact contained in A.

The set of facts has been provided with operators corresponding to ILL connectives (and therefore to LCC operators), we now translate formulas into facts.

Definition 4.3 *Given an enriched phase space, a* valuation *is a mapping η from atomic formulas to facts such that $\eta(\top) = \top$, $\eta(\mathbf{1}) = \mathbf{1}$ and $\eta(\mathbf{0}) = \mathbf{0}$.*

The interpretation $\eta(A)$ (resp. $\eta(\Gamma)$) of a formula A (resp. of a context Γ) is defined inductively in the obvious way:

$$\begin{array}{rcl}
\eta(A \otimes B) & = & \eta(A) \otimes \eta(B), \\
\eta(A \multimap B) & = & \eta(A) \multimap \eta(B), \\
\eta(!A) & = & !\eta(A), \\
\eta(A \,\&\, B) & = & \eta(A) \,\&\, \eta(B), \\
\eta(A \oplus B) & = & \eta(A) \oplus \eta(B), \\
\eta((\Gamma, \Delta)) & = & \eta(\Gamma) \otimes \eta(\Delta), \\
\eta(\forall x A) & = & \forall x \eta(A), \\
\eta(\exists x A) & = & \exists x \eta(A), \\
\eta(\Gamma) & = & \mathbf{1} \text{ if } \Gamma \text{ is empty.}
\end{array}$$

Sequents are interpreted as follows: $\eta(\Gamma \vdash A) = \eta(\Gamma) \multimap \eta(A)$. This brings one to defining a notion of validity:

Definition 4.4 (Validity) *Define:*
$\mathbf{P}$*,* $\eta \models (\Gamma \vdash A)$ *iff* $1 \in \eta(\Gamma \vdash A)$*, i.e.* $\eta(\Gamma) \subset \eta(A)$*,*
$\mathbf{P} \models (\Gamma \vdash A)$ *iff for every valuation* η*:* $\mathbf{P}$*,* $\eta \models (\Gamma \vdash A)$*,*
$\models (\Gamma \vdash A)$ *iff for every phase space* $\mathbf{P}$*:* $\mathbf{P} \models (\Gamma \vdash A)$*.*

This semantics of ILL formulas enjoys the following main properties:

Theorem 4.5 (Soundness [10, 25]) *If there is a sequent calculus proof of $\Gamma \vdash A$ then $\models (\Gamma \vdash A)$.*

Theorem 4.6 (Completeness [10, 25]) *If $\models (\Gamma \vdash A)$ then there is a sequent calculus proof of $\Gamma \vdash A$.*

4.2 Proving safety properties of LCC programs with the phase semantics

Using the phase semantics presented above we will now prove safety properties of CC programs. We use the soundness of the translation from LCC into ILL and so require, either to translate CC programs into LCC or to write programs directly in LCC.

The theorem 4.5 of soundness of the phase semantics w.r.t. ILL is:

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

It can easily be extended to $ILL_{\mathcal{C},\mathcal{D}}$ by imposing to any valuation η to satisfy the inclusions coming from the non-logical axioms (the axiom $c \vdash d$ imposes $\eta(c) \subset \eta(d)$).

By contrapositive we get:

$$\exists \mathbf{P}, \eta,\ \text{s.t. } \mathbf{P}, \eta \not\models (\Gamma \vdash A) \text{ implies } \Gamma \nvdash_{ILL_{\mathcal{C},\mathcal{D}}} A,$$

which is equivalent to:

$$\exists \mathbf{P}, \eta, \text{ s.t. } \eta(\Gamma) \not\subset \eta(A) \text{ implies } \Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A.$$

As the contrapositive of the theorem 3.7 of soundness from LCC to $ILL_{\mathcal{C},\mathcal{D}}$ is:

$$(X; c; \Gamma)^{\dagger} \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} (Y; d; \Delta)^{\dagger} \text{ implies } (X; c; \Gamma) \not\longrightarrow_{LCC} (Y; d; \Delta)$$

We have:

Proposition 4.7 *To prove a safety property of the kind:* $(X; c; \Gamma) \not\longrightarrow_{LCC} (Y; d; \Delta)$*, it is enough to show that:*
$\exists$ *a phase space* $\mathbf{P}$*, a valuation* η*, and an element* $a \in \eta((X; c; \Gamma)^{\dagger})$ *such that* $a \notin \eta((Y; d; \Delta)^{\dagger})$.

This proposition allows to reduce the problem of proving safety properties of CC programs, i.e. proving the non-existence of some derivation, to an existence problem: finding a phase structure, an interpretation and a counter-example for the above inclusion, or even, only proving their existence. Note that only soundness theorems are used, the second part of the correspondence (completeness) gives a certain certitude that when looking for a semantical proof of a true safety property, it exists!

4.3 Example 1 - Dining philosophers

As shown in section 2.2 the dining philosophers problem can easily be encoded in LCC. Let us try to prove, for instance, that this encoding satisfies the safety property that it does not allow two philosophers to eat with the same fork at the same time, independently of the number of philosophers.

i) Reformulating the property. We first have to express that we don't want two neighbors to eat together in a safety property of the above form:

$$\forall \mathtt{N}, \forall \mathtt{I}, \forall c, \forall A, (\emptyset; 1; \mathtt{init(N)}) \not\longrightarrow_{LCC} (\emptyset; \mathtt{eat(I)}, \mathtt{eat(I+1\ mod\ N)}, c; A)$$

From the corollary, it is enough to show:

$$\forall \mathtt{N}, \forall \mathtt{I}, \exists \mathbf{P}, \exists \eta, \exists x \in \eta(\mathtt{init(N)}), x \notin \eta(\mathtt{eat(I)} \otimes \mathtt{eat(I+1\ mod\ N)} \otimes \top)$$

where $\top$ is the usual constant of linear logic that means "anything" and so can replace c and A for any c and any A.

ii) Phase space. Consider the following structure $\mathbf{P}$:

- $\mathbb{N}$ (with it's usual product and unit) is the monoid,
- $\mathcal{F} = \mathcal{P}(\mathbb{N})$,
- $O = \{\emptyset, \{1\}\}$.

It is definitely a phase structure.

iii) Valuation. We need to define a valuation η on `fork(I)`, `eat(I,N)`, `N=M`, `N`$\neq$`M`, `philosopher(I,N)`, `recphilo(M,P)` and `init(N)`. We must not forget to check that the conditions coming from the declarations (non-logical axioms which translate into compulsory inclusions) of `philosopher(I,N)`, `recphilo(M,P)` and `init(N)` are satisfied.

Let us define η as follows:

$$\eta(\texttt{fork(I)}) = \{f_i\} \qquad \eta(\texttt{N = M}) = \begin{cases} \{1\} & \text{if } n = m, \\ \emptyset & \textit{otherwise}, \end{cases}$$
$$\eta(\texttt{eat(I,N)}) = \{e_{i,n}\}$$
$$\eta(\texttt{philosopher(I,N)}) = \{p_i\}$$
$$\eta(\texttt{recphilo(M,P)}) = \{x_{m,p} \cdot y_{m,p}\} \qquad \eta(\texttt{N} \neq \texttt{M}) = \begin{cases} \{1\} & \text{if } n \neq m, \\ \emptyset & \textit{otherwise}, \end{cases}$$
$$\eta(\texttt{init(N)}) = \{x_{1,n} \cdot y_{1,n}\}$$

where the indices (i, m, n, p) are the canonical interpretation of the corresponding integer variables, f_i and p_i are distinct prime numbers, and $e_i, x_{m,p}$ and $y_{m,p}$ are defined as follows:

$$e_{i,n} = f_i \cdot f_{i+1 \bmod n} \cdot p_i$$

$$x_{m,p} = \begin{cases} 1 & \text{if } m = p, \\ p_m \cdot f_m \cdot x_{m+1,p} \cdot y_{m+1,p} & \text{otherwise.} \end{cases}$$
$$y_{m,p} = \begin{cases} p_m \cdot f_m & \text{if } m = p, \\ 1 & \text{otherwise.} \end{cases}$$

The conditions coming from the declarations are:
– $\forall i, \{p_i\} \subset E_{i,n}$ where $E_{i,n} = \eta($body of `philosopher(I,N)`$)$
– $\forall m, \forall p, \{x_{m,p} \cdot y_{m,p}\} \subset F_{m,p}$ where $F_{m,p} = \eta($body of `recphilo(M,P)`$)$
– $\forall n, \{x_{1,n} \cdot y_{1,n}\} \subset \eta($body of `init(N)`$)$.

One can easily notice that the third condition is implied by the second one. Remembering that an agent $A \longrightarrow_{LCC} B \longrightarrow_{LCC} C \longrightarrow_{LCC} D$ is interpreted as $\{x \in \mathbb{N} : \forall y \in \eta(A), \forall z \in \eta(B), \forall t \in \eta(C), \exists u \in \eta(D), x \cdot y \cdot z \cdot t = u\}$, we can deduce: $E_{i,n} = \{x \in \mathbb{N} : \exists y \in G_{i,n}, f_{i+1 \bmod n} \cdot f_i \cdot x = e_{i,n} \cdot y\}$
with $G_{i,n} = \{y \in \mathbb{N} : e_{i,n} \cdot y = f_i \cdot f_{i+1 \bmod n} \cdot p_i\}$
(y represents an element of the interpretation of the part `eat(I,N)` $\rightarrow$ `(tell(fork(I)` $\otimes$ `fork(I+1 mod N))` $\|$ `philosopher(I,N))` in the $\texttt{I}^{th}$ philosopher).
Now, observe that the first condition $\forall i, \{p_i\} \subset E_{i,n}$ reduces to showing that $G_{i,n}$ is non-empty, which is true as $1 \in G_{i,n}$. The second condition on $F_{m,p}$ is verified with a simple induction on $x_{m,p}$ and $y_{m,p}$ which have been so defined on purpose. The valuation is thus correct.

iv) Counter-example. As $\eta(\texttt{init(N)}) = \{x_{1,n} \cdot y_{1,n}\}$ we must prove:
$x_{1,n} \cdot y_{1,n} \notin \eta(\texttt{eat(I,N)} \otimes \texttt{eat(I+1 mod N,N)} \otimes \top) =$
$\{x \in \mathbb{N} : \exists a \in \mathbb{N}, x = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a\}$.

First we show by induction that: $x_{1,n} \cdot y_{1,n} = f_1 \cdot \ldots \cdot f_n \cdot p_1 \cdot \ldots \cdot p_n$

And then proceed *Ad absurdum*:
if $x_{1,n} \cdot y_{1,n} = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a$ then

$$\begin{aligned} f_{i+1 \bmod n} \cdot x_{1,n} \cdot y_{1,n} &= e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a \cdot f_{i+1 \bmod n} \\ &= f_1 \cdot \ldots \cdot f_{i-1} \cdot f_{i+3} \cdot \ldots \cdot f_n \cdot \\ &\quad\; p_1 \cdot \ldots \cdot p_{i-1} \cdot p_{i+2} \cdot \ldots \cdot p_n \cdot e_{i,n} \cdot e_{i+1 \bmod n,n} \end{aligned}$$

hence, simplifying we get

$$\begin{aligned} a \cdot f_{i+1 \bmod n} &= f_1 \cdot \ldots \cdot f_{i-1} \cdot f_{i+3} \cdot \ldots \cdot f_n \cdot \\ & \quad p_1 \cdot \ldots \cdot p_{i-1} \cdot p_{i+2} \cdot \ldots \cdot p_n \end{aligned}$$

which is impossible (prime factors decomposition: $f_{i+1 \bmod n}$ appears on the left hand of $=$ but not on the right hand, a product of prime numbers), qed.

Remark:
– It is worth noting that, although a similar soundness theorem holds for the translation of agents into intuitionistic logic (IL), if we use that translation instead of ILL we will not be able to prove anything because from
`philosopher(I)` $\wedge$ `fork(I)` $\wedge$ `fork(I+1)` $\vdash$ `eat(I,N)` and
`philosopher(I+1)` $\wedge$ `fork(I+1)` $\wedge$ `fork(I+2)` $\vdash$ `eat(I+1,N)`we can infer
`philosopher(I)`$\wedge$`philosopher(I+1)`$\wedge$`fork(I)`$\wedge$`fork(I+1)`$\wedge$`fork(I+2)` $\vdash$ `eat(I,N)`$\wedge$`eat(I+1,N)`.
– The phase structure might seem unnatural, but in this case it can be simply considered as the free commutative monoid built on the atomic constraints of the program, interpreted as singletons, and generated by the equalities coming from the non-logical axioms (i.e. inclusions between singletons). Such a singleton-based phase structure cannot always be used however. For instance with the following program: `P = tell(d)`, `Q = c` $\rightarrow$ `P`, a singleton-based phase structure does not allow one to prove that `c` is not accessible from `P`, i.e. `P` $\not\vdash$ `c` $\otimes \top$, as we can deduce $\eta(\mathtt{P}) = \mathtt{c} \cdot \eta(\mathtt{Q})$ from the second declaration.

4.4 Example 2 - Producer/Consumer

The producer/consumer protocol with m producers and k consumers communicating via a buffer of size n can be encoded in LCC as follows:

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

Let us prove, with the same phase structure as above, that this protocol, encoded this way, is deadlock free, and safe (the number of units consumed is always less than the number of units produced).

4.4.1 Deadlock Freeness

The first task is to state this safety property in the form $(X; c; \Gamma) \not\longrightarrow_{LCC} (Y; d; \Delta)$. One can easily see that a deadlock may only occur if there is, either no `P` left, or no `C` left, or nothing to consume (`dem` and `pro`).

i) We thus want to prove $\mathtt{init} \not\longrightarrow_{LCC} \mathtt{dem}^{n'} \parallel \mathtt{P}^{m'} \parallel \mathtt{C}^{k'} \parallel \mathtt{pro}^{l'}$, with either $n' = l' = 0$ or $m' = 0$ or $k' = 0$.

ii) Let us now consider the structure $P = \mathbb{N}$, $\mathcal{F} = \mathcal{P}(\mathbb{N})$, and $O = \{\emptyset, \{1\}\}$, it is obviously a phase structure.

iii) Let us define the following valuation:

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

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

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

iv) Instead of exhibiting a counter-example, we will again prove *Ab absurdum* that the inclusion $\eta(\texttt{init}) \subset \eta(\texttt{dem}^{n'} \parallel \texttt{P}^{m'} \parallel \texttt{C}^{k'} \parallel \texttt{pro}^{l'})$ is impossible.

Suppose $\eta(\texttt{init}) \subset \{5^{n'} \cdot 2^{m'} \cdot 3^{k'} \cdot 5^{l'}\}$ Comparing the power of 5, 3 and 2, anything else than: $n' + l' = n$ and $m' = m$ and $k' = k$ is impossible, and therefore if there is a deadlock ($n' + l' = 0 \neq n$, or $m' = 0 \neq m$, or $k' = 0 \neq k$) $\eta(\texttt{init})$ is not a subset of its interpretation and thus init does not reduce into it, qed.

4.4.2 Safety

In order to check that there are never more units consumed than units produced, the encoding must be slightly modified to make this information directly observable:

```
P = dem → (pro || P || ∀X (np=X → np=X+1))
C = pro → (dem || C || ∀X (nc=X → nc=X+1))
init = dem^n || P^m || C^k || np=0 || nc=0
```

This kind of modification, namely adding an "oracle" to observe the property of interest, is commonly seen in other verification techniques, for instance when adding a separate automaton in model checking.

i) We want to prove:
$\texttt{init} \not\longrightarrow_{LCC} \texttt{dem}^{n'} \parallel \texttt{pro}^{l'} \parallel \texttt{P}^m \parallel \texttt{C}^k \parallel \texttt{np=np}_0 \parallel \texttt{nc=nc}_0$
with $\texttt{nc}_0 > \texttt{np}_0$.

ii) Once again we can use a quite simple structure, $P = \mathbb{Q}, \mathcal{F} = \mathcal{P}(\mathbb{Q}), O = \{\emptyset, \{1\}\}$.

iii) And the following valuation:

$$\eta(\texttt{dem}) = \{6\} \;\; \eta(\texttt{pro}) = \{3\} \;\; \eta(\texttt{P}) = \{5\} \;\; \eta(\texttt{C}) = \{7\}$$

$$\eta(\texttt{np=X}) = \{2^x\} \;\; \eta(\texttt{nc=X}) = \{2^{-x}\}$$

$$\eta(\texttt{init}) = \{2^n \cdot 3^n \cdot 5^m \cdot 7^k\}$$

This valuation is correct: $\exists d = 2$ s.t. $dem \cdot p = pro \cdot p \cdot d$ and $\forall x, 2^x \cdot d = 2^{x+1}$ hence $\eta(\texttt{P}) \subset \eta(\text{body of } \texttt{P})$.

iv) Now, It suffices to remark that $\texttt{nc}_0 > \texttt{np}_0$ and $\eta(\texttt{init}) \subset \{6^{n'} \cdot 3^{l'} \cdot 5^m \cdot 7^k \cdot 2^{np_0} \cdot 2^{-nc_0}\}$ would imply $l' < 0$, which is impossible, qed.

4.5 Example 3 - Mutual exclusion

There is no ";" (sequentiality operator) in the syntax, but it can be added without losing the soundness properties by translating it as "$\otimes$" (similarly, the guarded choice can be translated as " & "). The following example of mutual exclusion with semaphores shows an example of LCC + ";" program on which one can prove safety properties:

$$\begin{array}{l}
\mathtt{P}_i \ \mathtt{=}\ \mathtt{sem} \rightarrow \mathtt{cs} \parallel \mathtt{A}\ \mathtt{;}\ (\mathtt{cs} \rightarrow \mathtt{sem} \parallel \mathtt{P}_i) \\
\mathtt{init}\ \mathtt{=}\ \mathtt{sem} \parallel \mathtt{P}_1 \parallel \cdots \parallel \mathtt{P}_N
\end{array}$$

i) We want to prove that the two critical sections cs cannot take place at the same time: $\forall B, \mathtt{init} \not\longrightarrow_{LCC} \mathtt{cs} \parallel \mathtt{cs} \parallel B$ i.e. $\mathtt{init}^{\ddagger} \not\vdash \mathtt{cs} \otimes \mathtt{cs} \otimes \top$.

ii, iii, iv) The structure $P = \mathbb{N}$ and the valuation $\eta(\mathtt{sem}) = \eta(\mathtt{cs}) = \eta(\mathtt{init}) = \{2\}$ and $\eta(\mathtt{A}) = \eta(\mathtt{P}_i) = \{1\}$ are correct. The proof of existence of a counter-example (again *ab absurdum*) is trivial.

This handling of ";", or of the guarded choice, may be quite surprising and is of course not general. It shows that although these operators have no simple logical interpretation, it is nevertheless sometimes possible to capture their operational behavior in the statement of the property. In the previous example it was enough to show $\forall B, \mathtt{init} \not\longrightarrow_{LCC} \mathtt{cs} \parallel \mathtt{cs} \parallel B$ because we know that A is over when we remove cs from the store; showing $\forall B, \mathtt{init} \not\longrightarrow_{LCC} \mathtt{A} \parallel \mathtt{A} \parallel B$ would not be possible by interpreting ";" as "$\otimes$".

5 Conclusion and perspectives

Building upon the close correspondence between CC executions and proof search in LL, we have shown that the semantics of provability (not of proofs) in LL provides an interesting level of abstraction for reasoning about CC programs, getting rid of unnecessary execution details. In particular we have shown that various safety properties of simple protocol CC programs could be proved directly, simply by exhibiting a phase space and an interpretation of the program in which the property holds. This line of research has been continued in [39], where phase semantics is successfully applied to prove safety properties of other examples of CC programs.

These results open also a lot of questions, for instance on the shape of the simplest phase spaces for proving a given safety property, and on the possibility of automating such "semantical" proofs in a somewhat similar way to model checking. Preliminary results on counter phase model generation methods can be found in [26].

The method can be generalized to handle more safety properties of LCC programs. In particular the characterization of LCC suspensions [31] in the non-commutative logic of the second author [30], can be used to prove deadlock properties using non-commutative phase spaces.

The extension – induced by the logic – of CC languages to linear constraint systems is also interesting to study in its own right as it reconciles declarative programming with some form of imperative programming. We have shown this on simple examples for protocol specification. As another example, the rational reconstruction of CLP(FD) constraint propagators by CC agents given in [11] can be extended in LCC to cover the propagation algorithms of *global constraints* which use imperative (backtrackable) data structures [36]. Such a logical reconstruction with LCC of constraint solvers can thus be more faithful to current constraint programming practice.

Acknowledgments

We are particularly grateful to Dale Miller, Catuscia Palamidessi Vijay Saraswat and Rajeev Goré for fruitful discussions on this work, as well as to the anonymoous referees for their comments.

References

[1] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.

[3] E. Best, F.S. de Boer, and C. Palamidessi. Concurrent constraint programming with information removal. In *Proceedings of Coordination*, LNCS. Springer-Verlag, 1997.

[4] N. Carriero and D. Gelenter. Linda in context. *Comm. ACM*, 32(4):445–458, 1989.

[5] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM-TOPLAS*, 19(5):685–725, 1997.

[6] F. Fages. Constructive negation by pruning. *J. of Logic Programming*, 32(2), 1997.

[7] F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proc. 13th Annual IEEE Symposium on Logic in Computer Science, Indianapolis*, 1998.

[8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Model theoretic construction of the operational semantics of logic programs. *Information and Computation*, 102(1), 1993.

[9] J.H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, 1986.

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

[11] P. Van Hentenryck, V.A. Saraswat, and Y. Deville. Constraint processing in `cc(FD)`. Draft, 1991.

[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] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[14] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–581, May 1994.

[15] R. Jagadeesan, V. Shanbhogue, and V.A. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Parc, 1991.

[16] N. Kobayashi and A. Yonezawa. Logical, testing and observation equivalence for processes in a linear logic programming. Technical Report 93-4, Department of Computer Science, University of Tokyo, 1993.

[17] Y. Lafont and A. Scedrov. The undecidability of second order multiplicative linear logic. *Information and Computation*, 125(1):46–51, 1996.

[18] P. Lincoln, J. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56:239–311, 1992.

[19] P. Lincoln and V.A. Saraswat. Proofs as concurrent processes. Draft, 1991.

[20] P. Lincoln and N. Shankar. Proof search in first-order linear logic and other cut-free sequent calculi. In *Proc. 9th Annual IEEE Symposium on Logic in Computer Science, Paris*, 1994.

[21] M.J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of ICLP'87, International Conference on Logic Programming*, 1987.

[22] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, To appear, 2000. A preliminary version appeared in Proc. of the 1996 Workshop on Linear Logic available in Volume 3 of ENTCS.

[23] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1), 1992.

[25] M. Okada. Girard's phase semantics and a higher-order cut-elimination proof. Technical report, Institut de Mathématiques de Luminy, 1994.

[26] M. Okada and K. Terui. Completeness proofs for linear logic based on the proof search method (preliminary report). In J. Garrigue, editor, *Type theory and its applications to computer systems*, pages 57–75. Research Institute for Mathematical Sciences, Kyoto University, 1998.

[27] G. Perrier. Concurrent programming in linear logic. Technical Report CRIN 95-R-052, INRIA-Lorraine, 1995.

[28] A. Podelski and G. Smolka. Operational semantics of constraint logic programming with coroutining. In *Proceedings of ICLP'95, International Conference on Logic Programming*, Tokyo, 1995.

[29] P. Ruet. Logical semantics of concurrent constraint programming. In *Proc. CP'96*, 2^{nd} *Int. Conf. on Constraint Programming, Cambridge, MA*, LNCS 1118. Springer-Verlag, 1996.

[30] P. Ruet. *Logique non-commutative et programmation concurrente par contraintes.* PhD thesis, Université Denis Diderot, Paris 7, 1997.

[31] P. Ruet and F. Fages. Concurrent constraint programming and non-commutative logic. In *Proc. CSL'97, Annual Conf. EACSL*, LNCS 1414. Springer-Verlag, 1998.

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

[33] V.A. Saraswat, R. Jagadeesan, and V. Gupta. Defaut timed concurrent constraint programming. In *POPL'95: Proceedings ACM Symposium on Principles of Programming Languages*, 1995.

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

[35] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *POPL'91: Proceedings 18th ACM Symposium on Principles of Programming Languages*, 1991.

[36] V. Schächter. *Programmation concurrente avec contraintes fondée sur la logique linéaire.* PhD thesis, Université d'Orsay, Paris 11, 1999.

[37] P.J. Stuckey. Constructive negation for constraint logic programming. *Information and Computation*, 118(1), 1995.

[38] C. Tse. The design and implementation of an actor language based on linear logic. Master's thesis, MIT, 1994.

[39] Y. Yilma. Non-monotonic concurrent constraint programming verification using phase semantics. Master's thesis, School of Information Technology and Engineering, University of Ottawa, 1999.

A Appendix: Intuitionistic linear sequent calculus

Definition A.1 (Intuitionistic formulas) *The* intuitionistic formulas *are built from atoms* $p, q, \ldots$ *with:*

- *the multiplicative connectives:* $\otimes$ (tensor) *and implication* $\multimap$*,*
- *the additive connectives:* $\&$ (with) *and* $\oplus$ (plus),
- *the exponential connective* ! (of course *or* bang),
- *the constants: multiplicative* $\mathbf{1}$ *and* $\perp$*, and additive* $\top$ *and* $\mathbf{0}$*,*
- *the quantifiers: universal* $\forall$ *and existential* $\exists$*.*

Definition A.2 (Intuitionistic sequents) *The* sequents *are of the form* $\Gamma \vdash A$ *or* $\Gamma \vdash$*, where* A *is a formula and* Γ *is a multiset of formulas.*

The sequent calculus is given by the following rules:

Axiom - Cut

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

Multiplicatives

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

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

Additives

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

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

Constants

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

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

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

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