Phase Semantics and Verification of Concurrent Constraint Programs

François Fages Paul Ruet* Sylvain Soliman

CNRS - LIENS
Ecole Normale Supérieure†, Paris

Abstract

The class CC of concurrent constraint programming languages and its non-monotonic extension LCC based on linear constraint systems can be given a logical semantics in Girard's intuitionistic linear logic for a variety of observables. In this paper we settle basic completeness results and we show how the phase semantics of linear logic can be used to provide simple and very concise "semantical" proofs of safety properties for CC or LCC programs.

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 [10]. The class CLP($\mathcal{X}$) of constraint logic programming languages, introduced by Jaffar and Lassez [9], 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 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 [22] in 1987 arised as a natural combination of constraint logic programming and concurrent logic programming, with a synchronization mechanism based on constraint entailment [16]. 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($\mathbb{R}$) [9], 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.

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 [25, 11]. Such denotational semantics are used in [4] 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 paper, we explore another route based on Girard's intuitionistic linear logic (ILL) [6]. 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.

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. Linear constraint systems have been proposed in [24] in a higher-order setting which will not be considered here. LCC is a non-monotonic extension of CC, somewhat similar to [2] or [23], but where constraints are consumed by ask agents without dependency maintenance or recomputation. 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. Classical CC programs can however be recovered by the usual translation of intuitionistic logic into linear

*Current affiliation: McGill University (research supported by the INRIA).

†Contact author: François Fages, CNRS - LIENS, Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris (France), Email: fages@dmi.ens.fr.

logic [6].

Section 3 settles the basic soundness and completeness results of LCC operational semantics w.r.t. intuitionistic linear logic. Results similar to those of this section are part of the folklore on CC languages [15, 24] but have not been published, we refer to the appendix B and [5, 20] for complete proofs. 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.

Then we show in section 4 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 [17, 8] and focusing proofs [1], and further works on the design of concurrent languages based on proof search in LL [12, 19].

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.

Recently, phase semantics has gained interest in its applications to cut elimination [18], complexity of provability and decidability [13] (see [14] for a survey). Section 4 presents a new field of application of the phase semantics, yet unexplored though quite natural in the paradigm of concurrent logic programming.

2 Linear CC: operational semantics

2.1 Syntax

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

– $\mathcal{C}$ *is a set of formulas of intuitionistic linear logic (ILL), called 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* $\exists$ *and the exponential connective* $!$*;*

– $\Vdash_{\mathcal{C}}$ *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 rules of ILL for* 1*,* $\otimes$*,* $\exists$ *and* $!$ *(see appendix A).*

The classical constraint systems of CC languages [25] can be recovered with the usual translation of classical or intuitionistic logic into linear logic, by writing all constraints under a $!$ [6]. 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 2.2 *The* subsumption preorder $>$ *is defined by:* $c > d$ *iff* $c \vdash_{\mathcal{C}} d \otimes \top$.

Recall that $\top$ is the "true" connective of linear logic (see appendix A): therefore $c > d$ iff there exists a formula A s.t. $c \vdash_{\mathcal{C}} d \otimes A$.

LCC *agents* are defined by the grammar $A ::= p(\vec{x}) \mid tell(c) \mid (A \parallel A) \mid A + A \mid \exists x A \mid c \to A$ where $\parallel$ stands for parallel composition, $+$ for nondeterministic choice, $\exists$ for variable hiding and $\to$ for blocking ask. The atomic agents $p(\vec{x})\ldots$ are *procedure calls*. Recursion is obtained by *declarations* $D ::= \epsilon \mid p(\vec{x}) = A \mid D, D$. As usual, we assume that in a declaration $p(\vec{x}) = A$, all the free variables occurring in A have a free occurrence in p, and that in a relation $c_1 \ldots c_n \Vdash_{\mathcal{C}} c$, all the free variables occurring in c have a free occurrence in $c_1 \ldots c_n$. An LCC *program* is a couple $\mathcal{D}.A$ composed of a set $\mathcal{D}$ of declarations and an initial agent A. In the following, both the constraint system $\mathcal{C}$ and the set of declarations $\mathcal{D}$ will be implicit.

Definition 2.3 (Configurations) *A* configuration *is a triple* $(\vec{x}; c; \Gamma)$*, where* $\vec{x}$ *is a set of variables,* c *is a constraint, and* Γ *a multi-set of agents.*

In a configuration $(\vec{x}; c; \Gamma)$, $\vec{x}$ denotes the set of hidden variables and c is the store. We shall use the notation $\vec{x}, y$ to denote the union of the disjoint sets $\vec{x}$ and $\{y\}$, $fv(A)$ to denote the set of free variables of an agent or formula A, and $v(A)$ to denote the set of all variables occurring in A. The operational semantics of LCC programs can be presented by a congruence and a transition relation between configurations.

Definition 2.4 *The* structural congruence $\equiv$ *is the least congruence satisfying the rules of table 1.*

α-Conversion	$\dfrac{z \notin v(A)}{\exists y A \equiv \exists z A[z/y]}$
Parallel composition	$(\vec{x}; c; A \parallel B, \Gamma) \equiv (\vec{x}; c; A, B, \Gamma)$
Hiding	$\dfrac{y \notin \vec{x} \cup fv(c, \Gamma)}{(\vec{x}; c; \exists y A, \Gamma) \equiv (\vec{x}, y; c; A, \Gamma)} \quad \dfrac{y \notin fv(c, \Gamma)}{(\vec{x}, y; c; \Gamma) \equiv (\vec{x}; c; \Gamma)}$

Table 1: Structural congruence

Tell	$(\vec{x}; c; tell(d), \Gamma) \longrightarrow (\vec{x}; c \otimes d; \Gamma)$
Ask	$\dfrac{c \vdash_{\mathcal{C}} d \otimes e}{(\vec{x}; c; e \to A, \Gamma) \longrightarrow (\vec{x}; d; A, \Gamma)}$
Procedure calls	$\dfrac{(p(\vec{y}) = A) \in P}{(\vec{x}; c; p(\vec{y}), \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)}$
$\equiv$	$\dfrac{(\vec{x}; c; \Gamma) \equiv (\vec{x}'; c'; \Gamma') \qquad (\vec{x}'; c'; \Gamma') \longrightarrow (\vec{y}'; d'; \Delta') \qquad (\vec{y}'; d'; \Delta') \equiv (\vec{y}; d; \Delta)}{(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)}$
Blind choice	$(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma) \qquad (\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; B, \Gamma)$

Table 2: Transition relation

The transition *relation $\longrightarrow$ is the least transitive relation on configurations satisfying the rules of table 2.*

The reflexive closure of $\longrightarrow$ is denoted by $\longrightarrow^*$.

The only difference with monotonic CC is that constraints are formulas of linear logic and that the communication rule *ask* consumes information. In the *ask* rule, the resulting store d is the part of the store which remains after some necessary consumptions for entailing the guard e. Without restriction, LCC computation is thus intrinsically non-deterministic, even without the choice operator $+$, since several constraints can satisfy the condition of the *ask* rule. Of particular interest in this context are linear atomic constraints without entailment, since in that case, subsumption by a store is simply multiset inclusion, so a more practical deterministic *ask* rule with such constraints can be given by fixing $d = c \setminus e$. All the examples of communication protocol given in section 2.2 fall into this category.

Definition 2.5 (Observables) *The* store *of a configuration $(\vec{x}; c; \Gamma)$ is the constraint $\exists \vec{x} c$. An* accessible store *from A is a constraint d such that there exist a constraint c and a multi-set Γ of agents such that $(\emptyset; 1; A) \longrightarrow (\vec{x}; c; \Gamma)$ and $\exists \vec{x} c > d$.*

A success *for an agent A is a configuration $(\vec{x}; c; \emptyset)$ such that $(\emptyset; 1; A) \longrightarrow (\vec{x}; c; \emptyset)$;*

A suspension *for A is a configuration $(\vec{x}; c; d_1 \to A_1, \ldots, d_n \to A_n)$, $n \geq 0$, such that $(\emptyset; 1; A) \longrightarrow (\vec{x}; c; d_1 \to A_1, \ldots, d_n \to A_n)$ and for no i, $c > d_i$ (a success is a suspension with $n = 0$).*

An agent A suspends with the store c on the constraints $d_1, ..., d_n$, *if there exists a suspension for A of the form $(\vec{x}; c; d_1 \to A_1, \ldots, d_n \to A_n)$.*

Replacing the blind choice rule by the rules for the *one-step guarded choice*: $\dfrac{(\vec{x}; c; A, \Gamma) \longrightarrow (\vec{y}; d; \Delta)}{(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{y}; d; \Delta)}$ and $\dfrac{(\vec{x}; c; B, \Gamma) \longrightarrow (\vec{y}; d; \Delta)}{(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{y}; d; \Delta)}$ would obviously change the suspensions of a program. It is worth

noting however that the set of successes, as well as the set of accessible stores remain the same under both interpretations. Moreover as remarked in [11, 4] the difference between don't-know non-determinism (backtracking non-determinism) and don't-care non-determinism (committed-choice indeterminacy) arises in the way observations are defined, more than the way transitions are defined. For the sake of simplicity we have not explicitly treated here all forms of non-determinism, but the proof method presented in this paper remains valid as long as only accessible stores and successes are observed.

2.2 Example

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 of them has one fork on his (or her) right, and thus also one fork on his (or her) left, and needs these two forks to eat (the chop-sticks version may be more realistic).

As suggested in [2], the programming of this problem in LCC is very simple. 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 entailment relation is trivial. The code is described in table 3.

Note that unlike a classical CC program, the imperative data structures are encoded directly with linear constraints, instead of streams [22], and that unlike the Linda version of [3], 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 4.2, using the phase semantics of linear logic.

3 Logical semantics

In this section we interpret LCC agents as ILL formulas, and we settle soundness and completeness results of LCC computations w.r.t. ILL. These results hold for (monotonic) CC as well, since LCC is a conservative extension of CC. Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a linear constraint system and $\mathcal{D}$ be a set of declarations.

Definition 3.1 *LCC agents are translated into formulas in the following way:*

$$\begin{array}{ll} tell(c)^\dagger = c & (c \to A)^\dagger = c \multimap A^\dagger \\ p(\vec{x})^\dagger = p(\vec{x}) & (\exists x A)^\dagger = \exists x A^\dagger \\ (A \parallel B)^\dagger = A^\dagger \otimes B^\dagger & (A + B)^\dagger = A^\dagger \,\&\, B^\dagger \end{array}$$

If Γ *is the multi-set of agents* $(A_1 \dots A_n)$*, define* $\Gamma^\dagger = A_1^\dagger \otimes \cdots \otimes A_n^\dagger$*. If* $\Gamma = \emptyset$ *then* $\Gamma^\dagger = 1$.

The translation of a configuration $(\vec{x}; c; \Gamma)$ *is the formula* $\exists \vec{x}(c \otimes \Gamma^\dagger)$.

Let us denote by ILL($\mathcal{C}$,$\mathcal{D}$) 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 $p(\vec{x}) \vdash A^\dagger$ for every declaration $p(\vec{x}) = A$ in $\mathcal{D}$.

Theorem 3.2 (Soundness) *Let* $(\vec{x}; c; \Gamma)$ *and* $(\vec{y}; d; \Delta)$ *be LCC configurations.*

If $(\vec{x}; c; \Gamma) \equiv (\vec{y}; d; \Delta)$ *then* $(\vec{x}; c; \Gamma)^\dagger \dashv\vdash_{ILL(\mathcal{C},\mathcal{D})} (\vec{y}; d; \Delta)^\dagger$.

If $(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)$ *then* $(\vec{x}; c; \Gamma)^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} (\vec{y}; d; \Delta)^\dagger$.

Conversely, one can characterize the observation of successes, even in presence of the operator of explicit choice + (the proofs are given in the appendix B):

Theorem 3.3 (Observation of successes) *Let* A *be an agent LCC and* c *be a linear constraint.*

If $A^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c$*, then there exists a success* $(\vec{x}; d; \emptyset)$ *for* A *such that* $\exists \vec{x} d \vdash_{\mathcal{C}} c$.

Theorem 3.4 (Observation of stores [15]) *Let* A *be an LCC agent and* c *be a linear constraint. If* $A^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top$*, then* c *is a store accessible from* A.

It is worth noting that in this interpretation of the choice operator with the connective & , we can charaterize "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, can be characterized with the connective $\oplus$ of LL. To this end the operational semantics has to be redefined with a rewriting relation $\Longrightarrow$ acting now on complete frontiers of the derivation tree. Formally a frontier Φ is a multi-set of configurations (which keeps track of all the alternatives). The rule for blind choice becomes:

Blind choice:
$\langle (\vec{x}; c; A + B, \Gamma), \Phi \rangle \Longrightarrow \langle (\vec{x}; c; A, \Gamma), (\vec{x}; c; B, \Gamma), \Phi \rangle$.

All the other rules are modified accordingly to act on frontier configurations. Then the logical translation $.^\ddagger$ of agents, where non-determinism is expressed with $\oplus$:

$(A+B)^\ddagger = A^\ddagger \oplus B^\ddagger$, $\langle (\vec{x}; c; \Gamma), \Phi \rangle^\ddagger = \exists \vec{x}(c^\ddagger \otimes \Gamma^\ddagger) \oplus \Phi^\ddagger$,

leads to:

```
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)
```

Table 3: The dining philosophers

Theorem 3.5 (Soundness of frontier transitions) *Let Φ and Ψ be two frontiers. If $\Phi \Longrightarrow \Psi$ then $\Phi^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \Psi^\ddagger$.*

Theorem 3.6 (Observation of frontier stores) *Let A be an LCC agent and c be a linear constraint. If $A^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top$ then $\langle(\emptyset; 1; A)\rangle \Longrightarrow \langle(\vec{x}_1; d_1; \Gamma_1), \ldots, (\vec{x}_n; d_n; \Gamma_n)\rangle$ with $\forall j, \exists \vec{x}_j d_j \vdash_{\mathcal{C}} c$.*

Theorem 3.7 (Observation of frontier successes) *If $A^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c$, then $\langle(\emptyset; 1; A)\rangle \Longrightarrow \langle(\vec{x}_1; d_1; \emptyset), \ldots, (\vec{x}_n; d_n; \emptyset)\rangle$ with $\forall j, \exists \vec{x}_j d_j \vdash_{\mathcal{C}} c$.*

These completeness results show that ILL can be used to prove those *liveness properties* of LCC programs which are expressible as the existence of successes or accessible stores, i.e. properties that something good will happen, either on some branch ("may" properties) or on all branches ("must" properties). On the other hand, the soundness theorem can be used to prove *safety properties*, this is the purpose of the next section.

4 Phase semantics

4.1 Phase semantics of intuitionistic linear logic

Phase semantics is the natural provability semantics of linear logic [6]. It will also reveal 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). Nevertheless it is simpler to recall Okada's definition of the phase semantics for full intuitionistic LL [18] and to extend it to constants (**1**, **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, *such that:*

– $\mathcal{F}$ is closed under arbitrary intersection,

– for all $A \subset P$, for all $F \in \mathcal{F}$, $A \multimap F = \{x \in P : \forall a \in A, a \cdot x \in F\}$ is a fact.

As we shall see, facts correspond to ILL formulas and thus to LCC agents (cf. section 3).

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

A *parametrical 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 parametrical fact, and any operation defined on facts can be extended to parametrical facts: $(A \star B)(x) = A(x) \star B(x)$.

Let A, B be (parametrical) 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),

*– **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:

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

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 [6, 18]) *If there is a sequent calculus proof of $\Gamma \vdash A$ then $\models (\Gamma \vdash A)$.*

Theorem 4.6 (Completeness [6, 18]) *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

By using the phase semantics presented above, we 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 soundness theorem 4.5 of the phase semantics w.r.t. ILL, $\Gamma \vdash_{ILL} A$ implies $\forall P,\ \eta, P, \eta \models (\Gamma \vdash A)$, can be easily extended to $ILL_{\mathcal{C},\mathcal{D}}$, by requiring valuations η to satisfy the inclusions coming from the non-logical axioms (the axiom $c \vdash d$ imposes $\eta(c) \subset \eta(d)$ and the axiom $p(\vec{y}) \vdash A$ imposes $\eta(p(\vec{y})) \subset \eta(A)$). Now when we talk about *valuations*, we mean valuations satisfying the inclusions coming from $(\mathcal{C}, \mathcal{D})$. By contrapositive we get: $\exists P, \eta$, s.t. $P, \eta \not\models (\Gamma \vdash A)$ implies $\Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A$, which is equivalent to: $\exists P, \eta$, s.t. $\eta(\Gamma) \not\subset \eta(A)$ implies $\Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A$. Now by contrapositive of the soundness theorem 3.2: $(\vec{x}; c; \Gamma)^\dagger \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} (\vec{y}; d; \Delta)^\dagger$ implies $(\vec{x}; c; \Gamma) \not\longmapsto (\vec{y}; d; \Delta)$, we get:

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

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 space, 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 actually 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 do not want two neighbors to eat together in a safety property of the above form:

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

By the above property, it is enough to show:

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

where $\top$ is the usual constant of linear logic that means "anything", and stands here for $c \otimes A$ for any constraint c and any A.

ii) Phase space. Consider the following structure P:
– the monoid is $\mathbb{N}$ (with its usual product and unit),
– $\mathcal{F} = \mathcal{P}(\mathbb{N})$,
– $O = \{\emptyset, \{1\}\}$. It is definitely a phase space.

iii) Valuation. We need to define a valuation η on `fork(I)`, `eat(I,N)`, `N=M`, `N≠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:

$$\begin{aligned}
\eta(\texttt{fork(I)}) &= \{f_i\} \\
\eta(\texttt{eat(I,N)}) &= \{e_{i,n}\} \\
\eta(\texttt{philosopher(I,N)}) &= \{p_i\} \\
\eta(\texttt{recphilo(M,P)}) &= \{x_{m,p} \cdot y_{m,p}\} \\
\eta(\texttt{init(N)}) &= \{x_{1,n} \cdot y_{1,n}\}
\end{aligned}$$

$$\eta(\texttt{N = M}) = \begin{cases} \{1\} & \text{if } n = m, \\ \emptyset & \textit{otherwise.} \end{cases}$$

$$\eta(\texttt{N} \neq \texttt{M}) = \begin{cases} \{1\} & \text{if } n \neq m, \\ \emptyset & \textit{otherwise.} \end{cases}$$

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, $e_{i,n} = f_i \cdot f_{i+1 \bmod n} \cdot p_i$, and $x_{m,p}$ and $y_{m,p}$ are defined as follows:

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

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

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)`).

The third condition is implied by the second one. An agent $c \to d \to e \to A$ is interpreted as $\{x \in \mathbb{N} : \forall y \in \eta(c), \forall z \in \eta(d), \forall t \in \eta(e), \exists u \in \eta(A), x \cdot y \cdot z \cdot t = u\}$, hence: $E_{i,n} = \{x \in \mathbb{N} : \exists y \in G_{i,n}, f_{i+1 \bmod n} \cdot f_i \cdot x = e_{i,n} \cdot y\}$ where $G_{i,n} = \{y \in \mathbb{N} : e_{i,n} \cdot y = f_i \cdot f_{i+1 \bmod n} \cdot p_i\}$. ($y$ represents an element of the interpretation of the part `eat(I,N)` $\to$ `(tell(fork(I)` $\otimes$`fork(I+1 mod N))` $\parallel$ `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 show 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 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:
$x_{1,n} \cdot y_{1,n} \notin \eta(\texttt{eat(I,N)} \otimes \texttt{eat(I+1 mod N,N)} \otimes \top) = \{x \in \mathbb{N} : \exists a \in \mathbb{N}, x = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a\}$.

We first 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 *ab absurdum*: if $x_{1,n} \cdot y_{1,n} = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a$ then $f_{i+1 \bmod n} \cdot x_{1,n} \cdot y_{1,n} = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a \cdot f_{i+1 \bmod n} = f_1 \cdot \ldots \cdot f_{i-1} \cdot f_{i+3} \cdot \ldots \cdot f_n \cdot p_1 \cdot \ldots \cdot p_{i-1} \cdot p_{i+2} \cdot \ldots \cdot p_n \cdot e_{i,n} \cdot e_{i+1 \bmod n,n}$.

Simplying we get $a \cdot f_{i+1 \bmod n} = f_1 \cdot \ldots \cdot f_{i-1} \cdot f_{i+3} \cdot \ldots \cdot f_n \cdot p_1 \cdot \ldots \cdot p_{i-1} \cdot p_{i+2} \cdot \ldots \cdot p_n$ 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.

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 space 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 equations coming from the non-logical axioms (i.e. inclusions between singletons). Such a singleton-based phase space cannot always be used however. For instance with the following program: `P = tell(d)`, `Q =` `c` $\to$ `P`, a singleton-based phase structure would not allow to prove that `P` never has `c` in his store, i.e. `P` $\not\vdash$ `c` $\otimes \top$, as we can deduce $\eta(\texttt{P}) = \texttt{c} \cdot \eta(\texttt{Q})$ 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:

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

Let us prove, with the same phase space 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 $(\vec{x}; c; \Gamma) \not\longmapsto (\vec{y}; d; \Delta)$. 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 $\mathtt{init} \not\longmapsto \mathtt{dem}^{n'} \parallel \mathtt{P}^{m'} \parallel \mathtt{C}^{k'} \parallel \mathtt{pro}^{l'}$, with either $n' = l' = 0$ or $m' = 0$ or $k' = 0$.

ii) The structure defined by $P = \mathbb{N}$, $\mathcal{F} = \mathcal{P}(\mathbb{N})$, and $O = \{\emptyset, \{1\}\}$ is obviously a phase space.

iii) Let us define the following valuation: $\eta(\mathtt{dem}) = \{5\}$ $\eta(\mathtt{pro}) = \{5\}$ $\eta(\mathtt{P}) = \{2\}$ $\eta(\mathtt{C}) = \{3\}$ $\eta(\mathtt{init}) = \{2^m \cdot 3^k \cdot 5^n\}$.

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

iv) Instead of exhibiting a counter-example, we will again prove *ab absurdum* that the inclusion $\eta(\mathtt{init}) \subset \eta(\mathtt{dem}^{n'} \parallel \mathtt{P}^{m'} \parallel \mathtt{C}^{k'} \parallel \mathtt{pro}^{l'})$ is impossible. Assume $\eta(\mathtt{init}) \subset \{5^{n'} \cdot 2^{m'} \cdot 3^{k'} \cdot 5^{l'}\}$ and compare the exponents of 5, 3 and 2: anything else than $n' + l' = n$ and $m' = m$ and $k' = k$ is impossible, and therefore a deadlock ($n' + l' = 0 \neq n$, or $m' = 0 \neq m$, or $k' = 0 \neq k$) means that $\eta(\mathtt{init})$ 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 ⅋ np=X ⊸ np=X+1)
C = pro ⊸ (dem ⅋ C ⅋ 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 usual in other verification techniques, for instance when adding a separate automaton in model checking.

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

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:

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

$$\eta(\mathtt{np{=}X}) = \{2^x\} \quad \eta(\mathtt{nc{=}X}) = \{2^{-x}\}$$

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

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

iv) Now it suffices to remark that $\mathtt{nc}_0 > \mathtt{np}_0$ and $\eta(\mathtt{init}) \subset \{6^{n'} \cdot 3^{l'} \cdot 5^m \cdot 7^k \cdot 2^{np_0} \cdot 2^{-nc_0}\}$ 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:

```
P_i = sem ⊸ cs ⅋ (A ; (cs ⊸ sem ⅋ P_i))
init = sem ⅋ P_1 ⅋ ··· ⅋ P_N
```

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

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 way of handling ";", or 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 $\forall B, \mathtt{init} \not\longmapsto \mathtt{cs} \parallel \mathtt{cs} \parallel B$ because we know that A is over when we remove cs from the store; showing $\forall B, \mathtt{init} \not\longmapsto \mathtt{A} \parallel \mathtt{A} \parallel B$ 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.

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 automatizing such "semantical" proofs in a somewhat similar way to model checking. These issues should serve further investigation.

The method can be generalized to handle more safety properties of LCC programs. In particular the characterization of LCC suspensions [21] in the non-commutative logic of the second author [20],

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 rationale reconstruction of CLP(FD) constraint propagators by CC agents given in [7] could be extended in LCC to cover the propagation algorithms of *global constraints* which use imperative (backtrackable) data structures. Such a logical reconstruction with LCC would be thus more faithful to current CLP(FD) practice.

References

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

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

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

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

[5] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. Technical report, DMI - LIENS, In preparation.

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

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

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

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

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

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

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

[21] P. Ruet and F. Fages. Concurrent constraint programming and non-commutative linear logic. In *Computer Science Logic'97*, LNCS. Springer-Verlag, 1998.

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

[23] 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.

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

[25] 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.

A Appendix: Intuitionistic linear sequent calculus

Definition A.1 (Intuitionistic formulas) *The* intuitionistic formulas *are built from atoms p, q, ... 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 **1** *and* $\bot$*, and additive* $\top$ *and* **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 multi-set of formulas.*

The sequent calculus is given by the following rules:

Axiom - Cut

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

Multiplicatives

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

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

Additives

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

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

Constants

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

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

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

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

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

B Appendix: Proofs of completeness theorems

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

- if ϕ is a constraint: "there exists a configuration $(\vec{y}; d; \emptyset)$, such that $\exists \vec{y} d \vdash_C \phi$ and $\kappa \longrightarrow^* (\vec{y}; d; \emptyset)$",
- if ϕ is a procedure name: "there exists a configuration $(\vec{y}; d; \phi)$, such that $fv(\phi) \cap \vec{y} = \emptyset$, $\exists \vec{y} d \vdash_C 1$ and $\kappa \longrightarrow^* (\vec{y}; d; \phi)$".

Lemma B.1 *Let* κ *and* λ *be two configurations LCC such that* $\kappa^\dagger = \lambda^\dagger$*, and* ϕ *be a constraint or a procedure name.*
$\kappa \xrightarrow{\gg} \phi$ *iff* $\lambda \xrightarrow{\gg} \phi$*.*

Proof. One shows by induction on the formula $\kappa^\dagger = \lambda^\dagger$ that $\kappa \xrightarrow{\gg} \phi$ iff $\lambda \xrightarrow{\gg} \phi$.

- If $\kappa^\dagger = \lambda^\dagger$ is atomic, it is clear.
- If $\kappa^\dagger = \lambda^\dagger = c \Rightarrow A^\dagger$, with c a constraint and A an agent, then κ and λ are necessarily both equal to $(\emptyset; 1; c \to A)$.
- If $\kappa^\dagger = \lambda^\dagger = \exists x A^\dagger$, then the only two possibilities for κ and λ are $(x, \vec{y}; c; \Gamma)$ and $(\emptyset; 1; \exists x \exists \vec{y}(tell(c) \parallel \Gamma))$, and these two configurations are structurally congruent, thus in all cases $\kappa \xrightarrow{\gg} \phi$ iff $\lambda \xrightarrow{\gg} \phi$.

- If $\kappa^\dagger = \lambda^\dagger = A^\dagger \otimes B^\dagger$, 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 $(\emptyset; c \otimes d; \emptyset)$ (if $A^\dagger = c$ and $B^\dagger = d$, constraints). The induction is useful only in the first case, and the result is evident.

□

Lemma B.2 *Let $(\emptyset; 1; \Gamma)$ be an LCC configuration such that $(\emptyset; 1; \Gamma) \stackrel{\ggg}{\longrightarrow} \phi$ and x be a variable such that $x \notin fv(\phi)$. Then $(x; 1; \Gamma) \stackrel{\ggg}{\longrightarrow} \phi$*

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

If ϕ is a constraint then $(\emptyset; 1; \Delta) \longrightarrow^* (\vec{y}; d; \emptyset)$ with $\exists \vec{y} d \vdash_{\mathcal{C}} \phi$. One can then show by an easy induction on this derivation that $(x; 1; \Delta) \longrightarrow^* (x, \vec{y}; d; \emptyset)$ (case analysis, the only non-trivial one 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 $(x, \vec{y}; d; \emptyset)$). As $x \notin fv(\phi)$ we get $\exists x, \vec{y}\ d \vdash_{\mathcal{C}} \phi$ so $(x; 1; \Delta) \stackrel{\ggg}{\longrightarrow} \phi$. If ϕ is a procedure name we also get $(x; 1; \Delta) \stackrel{\ggg}{\longrightarrow} \phi$ in the same manner. The result then follows from the observation that $(x; 1; \Delta) \equiv (x; 1; \Gamma)$. □

Lemma B.3 *Let $\kappa = (\vec{x}; c; \Gamma)$ be an LCC configuration, and ϕ be a constraint or a procedure name.*

If $\kappa^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \phi$, then $\kappa \stackrel{\ggg}{\longrightarrow} \phi$.

Proof. One proves the result for multisets of agents: if ${A_1}^\dagger, \ldots, {A_n}^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \phi$, then $(\emptyset; 1; A_1, \ldots, A_n) \stackrel{\ggg}{\longrightarrow} \phi$.

It allows one to conclude: indeed let $(\vec{x}; c; \Gamma)$ be a configuration, and ϕ be a constraint or a procedure name. Note that $(\vec{x}; c; \Gamma)^\dagger = \exists \vec{x}(tell(c) \parallel \Gamma)^\dagger$. Therefore if $(\vec{x}; c; \Gamma)^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} \phi$, then $(\vec{x}; 1; tell(c), \Gamma)) \equiv (\emptyset; 1; \exists \vec{x}(tell(c) \parallel \Gamma)) \stackrel{\ggg}{\longrightarrow} \phi$. But $(\vec{x}; 1; tell(c), \Gamma))^\dagger = (\vec{x}; c; \Gamma))^\dagger$. So according to Lemma B.1, $(\vec{x}; c; \Gamma)) \stackrel{\ggg}{\longrightarrow} \phi$, qed.

One proceeds by induction on a sequent calculus proof of $A_1^\dagger, \ldots, A_n^\dagger \vdash \phi$.

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 *declarations* for an axiom $p \vdash q$; the case of an axiom $d \vdash_{\mathcal{C}} e$ is trivial.

- π ends with a cut. Let us consider for instance :

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

  By induction hypothesis, $(\emptyset; 1; \Gamma) \stackrel{\ggg}{\longrightarrow} p$, i.e. there exists a configuration $(\vec{y}; d; p)$ such that $fv(p) \cap \vec{y} = \emptyset$, $\exists y d \vdash_{\mathcal{C}} 1$ and $(\emptyset; 1; \Gamma) \longrightarrow^* (\vec{y}; d; p)$. Thus $(\emptyset; 1; \Gamma) \longrightarrow^* (\vec{y}; d; \phi)$; $fv(\phi) \cap \vec{y} = \emptyset$ as $fv(\phi) \subset fv(p)$. If $\phi^\dagger = \phi$ is a procedure name, it is finished. If $\phi^\dagger$ is a constraint c, then $\phi = tell(c)$ whence $(\emptyset; 1; \Gamma) \longrightarrow^* (\vec{y}; d \otimes c; \emptyset)$ and $d \otimes c \vdash_{\mathcal{C}} c$, qed.

  The other cases are similar.

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

- π ends with:

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

  By induction hypothesis, $(\emptyset; 1; A, B, \Gamma) \stackrel{\ggg}{\longrightarrow} \phi$. If $A^\dagger$ and $B^\dagger$ are constraints, there is a priori an ambiguity on the agent C whose translation is $A^\dagger \otimes B^\dagger$: 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 B.1, $(\emptyset; 1; C, \Gamma) \stackrel{\ggg}{\longrightarrow} \phi$.

- π ends with:

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

  $\phi \otimes \psi$ 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 $(\emptyset; 1; \Gamma) \stackrel{\ggg}{\longrightarrow} \phi$ and $(\emptyset; 1; \Delta) \stackrel{\ggg}{\longrightarrow} \psi$ end to end.

- π ends with:

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

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

- 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, $(\emptyset; 1; A, \Gamma) \xrightarrow{\ggg} \phi$. 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 B.2, $(\emptyset; 1; \exists x A, \Gamma) \xrightarrow{\ggg} \phi$, qed.

- π ends with:

$$\frac{\Gamma^\dagger, A^\dagger \vdash \phi \quad \Delta^\dagger \vdash c}{\Gamma^\dagger, \Delta^\dagger, c \multimap A^\dagger \vdash \phi}$$

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

- π ends with a dereliction:

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

  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:

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

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

- π ends with a contraction:

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

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

□

Theorem 3.3 (Observation of successes) *Let A be an agent LCC and c be a linear constraint.*

If $A^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c$, then there exists a success for A $(\vec{x}; d; \emptyset)$ such that $\exists \vec{x} d \vdash_{\mathcal{C}} c$.

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

Recall that $\top$ is the additive true constant [6], which is neutral for &.

Theorem 3.4 (Observation of stores) *Let A be an LCC agent and c be a linear constraint. If $A^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top$, then c is a store accessible from A.*

Proof. One shows by easy induction on a proof of $\Gamma \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top$, where Γ is an arbitrary multi-set of formulas which are translations of agents $A_1, \ldots, A_n$, that c is a success for $(\emptyset; 1; A_1, \ldots, A_n)$. The proof uses Theorem 3.3, for the right introduction of the tensor connective in $c \otimes \top$. □

Theorems 3.6 and 3.7 (Observation of frontier stores and successes) *Let A be an LCC agent and c be a linear constraint.*

If $A^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top$ then $\langle(\emptyset; 1; A)\rangle \Longrightarrow \langle(\vec{x}_1; d_1; \Gamma_1), \ldots, (\emptyset \vec{x}_n; d_n; \Gamma_n)\rangle$ with $\forall j, \exists \vec{x}_j d_j \vdash_{\mathcal{C}} c$.

If $A^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c$ then $\langle(\emptyset; 1; A)\rangle \Longrightarrow \langle(\vec{x}; d_1; \emptyset), \ldots, (\vec{x}_n; d_n; \emptyset)\rangle$ with $\forall j, \exists \vec{x}_j d_j \vdash_{\mathcal{C}} c$.

Proof. We first prove the result for successes, for stores we apply the same argument as in the proof of theorem 3.4. Proceed by induction on a proof of $A^\ddagger \vdash_{ILL(\mathcal{C},\mathcal{D})} c$ as in lemma B.3. The only difference is the following induction step:

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

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