Logical semantics of concurrent constraint programming

Paul Ruet[1,2]

[1] LIENS, Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
[2] Thomson - LCR, Domaine de Corbeville, 91404 Orsay, France

Email: ruet@dmi.ens.fr

Abstract. This paper investigates logical characterizations of some aspects of concurrent constraint (`cc`) computations. It contains both negative and positive results.
We show that intuitionistic logic enables to observe the so-called *stores* of a concurrent constraint agent, but neither its successes nor its suspensions, even in the monotonic and deterministic case. On the other hand, IMALL (intuitionistic multiplicative and additive linear logic) does enable the observation of successes (but not that of suspensions): we consider a non-monotonic and non-deterministic version of `cc`, `lcc`, and we show that the successes of an `lcc` computation can be characterized logically; this holds also for `cc`, since `cc` can be faithfully translated into `lcc`.

Keywords
Concurrent constraint programming, intuitionistic logic, linear logic.

1 Introduction

Concurrent constraint programming `cc` [21] is a model of concurrent computation, where concurrent 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 a constraint c to the store, 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. Computation is *monotonic* (the constraints in the store are not consumed): this allows to provide `cc` with a denotational semantics, viewing agents as closure operators on the semi-lattice of constraints [23].

Syntactically, concurrent constraint programming is an extention of *constraint logic programming* [9, 14] with a suspension mechanism $c \rightarrow A$, and the operational semantics of `cc` is the same as that of constraint logic programming, except for $c \rightarrow A$ which blocks until the amount of accumulated information (the store) is strong enough to entail c (in intuitionistic or classical logic), in which case $c \rightarrow A$ evolves to A.

Besides, Saraswat and Lincoln have proposed a *non-monotonic* version of `cc` [22], further studied in [4, 25] where the logic of constraints is linear logic [8]: in this version, constraints can be consumed, and the language is therefore closer to process calculi like Milner's π-calculus [17].

In constraint logic programming (with, or without negation by failure, or constructive negation), the logical nature of the constraint system extends to the goals and program declarations, and states strong connections between operational semantics and entailment (in classical logic or 3-valued logic) [14, 12, 24, 6]. For instance, success constraints (i.e. final states of computations) can be observed logically: any success entails the initial state (modulo the completed 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 successes $c_1 \dots c_n$, i.e. $\mathcal{C} \vdash \forall(c_1 \dots c_n \Rightarrow c)$. Such results make easier the design and understanding of programs, and provide useful tools for reasoning about them.

In concurrent constraint programming, the situation is less clear. In [13] Lincoln and Saraswat give an interesting connection between the observation of the stores of `cc` agents and entailment in intuitionistic logic. However it tells just part of the story of a `cc` computation: for instance, it does not say anything about eventual suspending agents. Actually let a *success* of an agent A be a store c such that A evolves to c, and let a *suspension* be an agent $B = c \wedge (d \rightarrow A)$ such that A evolves to B and c does not entail d (the exact definition is slightly longer): in section 3 we shall show, through counter-examples, that the observation of successes or suspensions is not expressible in intuitionistic logic. Roughly speaking, the interpretation of `cc` agents as intuitionistic formulas stumbles against the structural rule of (left) *weakening*:

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

Girard's linear logic is a fine proof-theoretical study of the weakening and *contraction* structural rules of classical and intuitionistic logics. While moving to linear logic, it is very natural to move to a non-monotonic version of `cc` at the same time. This has been done by Saraswat and Lincoln in a higher-order setting. Here we define a first-order non-monotonic variant, `lcc`, for which we distinguish successes and suspensions (section 4), and we prove a completeness result on successes: we show that the successes of an `lcc` computation can be characterized in IMALL (intuitionistic multiplicative and additive linear logic) (section 5), whereas the suspensions cannot (section 6). We show that `cc` can be faithfully translated into `lcc`, so this result holds also for `cc`.

Finally we discuss in section 6 the limits of the correspondence between linear logic and `lcc` computations, and its signficance for linear logic and concurrency.

2 Monotonic cc

A *monotonic 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, and logical operators $\top$ (*true*), $\wedge$ and $\exists$; and $\Vdash_{\mathcal{C}} \subseteq \mathcal{C}^\star \times \mathcal{C}$. We assume $\wedge$ has neutral $\top$. Instead of $((c_1 \dots c_n), c) \in \Vdash_{\mathcal{C}}$, we write $c_1 \dots c_n \Vdash_{\mathcal{C}} c$.

$\vdash_{\mathcal{C}}$ is the least reflexive relation $\subseteq \mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the following rules of intuitionistic logic:

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

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

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

The set $\mathcal{A}$ of cc agents is given by the following grammar:

$$A ::= c \mid c \rightarrow A \mid A \wedge A \mid A \vee A \mid \exists x A \mid p(\boldsymbol{x})$$

where c stands for a constraint, $\wedge$ for parallel composition, $\vee$ for non-determinism and $\exists x$ for hiding of a variable x. In an agent $A = c \wedge A_1 \wedge \dots \wedge A_n$, if c is a constraint, the main constructor of each A_i is not $\wedge$, and no A_i is a constraint, we call c the *store* of A. It is the 'constraint' part of A.

Recursion is obtained with declarations:

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

The operational semantics is given in the style of the Chemical Abstract Machine [3] (see also [19]). This presentation, though different from a logic programming one, has the advantage of keeping track of the variable bindings, and we find it therefore cleaner to manage logically.

• The *structural congruence* $\equiv$ is the least congruent equivalence such that $(\mathcal{A}/\equiv, \wedge, \top)$ is an abelian monoid, $(\mathcal{A}/\equiv, \vee)$ is an abelian semi-group, $\wedge$ and $\vee$ are distributive with respect to each other, and such that, for all agents A and B:

$$\exists x \top \equiv \top \qquad \exists x \exists y A \equiv \exists y \exists x A$$

$$\frac{A \text{ and } B \text{ are } \alpha\text{-convertible}}{A \equiv B} \qquad \frac{x \text{ is not free in } A}{\exists x (A \wedge B) \equiv A \wedge \exists x B}$$

• The *transition* relation $\longrightarrow$ is the least reflexive transitive congruence such that:

$$c \wedge (c \to A) \longrightarrow c \wedge A \qquad \frac{c \vdash_{\mathcal{C}} d}{c \longrightarrow d} \qquad \frac{(p(\boldsymbol{x}) = A) \in P}{p(\boldsymbol{x}) \longrightarrow A}$$

$$\frac{A' \equiv A \qquad A \longrightarrow B \qquad B \equiv B'}{A' \longrightarrow B'} \qquad \frac{A \longrightarrow C \qquad B \longrightarrow C}{A \vee B \longrightarrow C}$$

Remarks:

▶ This version of cc is monotonic in the following sense: for any transition $c \wedge A \longrightarrow d \wedge B$ between agents $c \wedge A$ and $d \wedge B$ with respective stores c and d, there exists a transition $c \wedge A \longrightarrow c \wedge d \wedge B$ (even though d might not entail c, as constraints may vanish because of the second rule for the transition relation).

▶ The non-deterministic construct $\vee$ is not the angelic non-deterministic construct of [10], it is close to that of Lincoln and Saraswat in [13]: intuitively $A \vee B$ can be either A or B, but you cannot decide, your vision is 'blurred', so to be able to do a 'sharp' observation on the rest of the computation, both possibilities must have some common result C (a 'coincidence'); it is a form of hiding at the level of agents, in the same way as $\exists$ is a hiding of variables.

Agents and declarations not involving $\vee$ are said *deterministic*.

▶ Concurrent constraint programming languages are parameterized by a constraint system, which is often not mentioned explicitly in the operational semantics as well as in the constraint entailment. Declarations and constraint entailment are also implicit in the operational semantics, but the context will not allow any confusion.

Examples:

▶ $c \wedge (d \to A \vee e \to A) \equiv (c \wedge d \to A) \vee (c \wedge e \to A)$ (distributivity)
$\longrightarrow (d \wedge d \to A) \vee (e \wedge e \to A)$ if $c \vdash_{\mathcal{C}} d$ and $c \vdash_{\mathcal{C}} e$
$\longrightarrow (d \wedge A) \vee (e \wedge A)$
$\longrightarrow (\top \wedge A) \vee (\top \wedge A)$
$\equiv A \vee A \equiv A$ (transition rule for $\vee$)

▶ $c(x) \wedge \exists x(c(x) \to c(x))$ suspends because $\exists x(c(x) \to c(x))$ is not a constraint and therefore cannot be erased; whereas:

$c(x) \wedge \exists x(\top \to c(x)) \equiv c(x) \wedge \exists x(\top \wedge \top \to c(x))$ ($\top$ neutral for $\wedge$)
$\longrightarrow c(x) \wedge \exists x(\top \wedge c(x)) \equiv c(x) \wedge \exists x(c(x)) \longrightarrow c(x)$ ($\exists x(c(x))$ is a constraint)

3 Observations with intuitionistic logic

3.1 Monotonic stores

In concurrent constraint programming, the permanent information is expressed by the store, therefore the stores are very natural observations on cc agents. Indeed this is the approach of Lincoln and Saraswat [13]: the stores of monotonic cc agents can be observed with intuitionistic logic IL.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a monotonic constraint system, and $\mathcal{D}$ a set of declarations. Let IL($\mathcal{C}$,$\mathcal{D}$) be the deduction system obtained by extending IL with:

– elements of $\Vdash_{\mathcal{C}}$ as non-logical axioms,
– for each declaration $p(\boldsymbol{x}) = A$ in $\mathcal{D}$, the sequent $p(\boldsymbol{x}) \vdash A$ as non-logical axiom.

$\dashv$ denotes the inverse of $\vdash$ and $A \dashv\vdash B$ stands for $A \dashv B$ and $A \vdash B$.

Theorem 1 (Soundness) *Let A and B be* `cc` *agents. If $A \equiv B$ then $A \dashv\vdash_{IL(\mathcal{C},\mathcal{D})} B$. If $A \longrightarrow B$ then $A \vdash_{IL(\mathcal{C},\mathcal{D})} B$.*

Proof Trivial induction on $\equiv$ and $\longrightarrow$. ∎

Theorem 2 (Observation of monotonic stores) *Let A be a* `cc` *agent and c a constraint. If $A \vdash_{IL(\mathcal{C},\mathcal{D})} c$ then there are constraints $c_1 \dots c_n$ and agents $B_1 \dots B_n$ such that $A \longrightarrow (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$ and for all i, $c_i \vdash_{\mathcal{C}} c$.*

Sketch of proof It is simpler to prove the result for multisets of agents $A_1 \dots A_n$: if $A_1 \dots A_n \vdash c$ in IL($\mathcal{C}$,$\mathcal{D}$), then $A_1 \wedge \dots \wedge A_n \longrightarrow c$. We proceed by induction on a (sequent calculus) proof of $A_1 \dots A_n \vdash c$: remark that this works because the formula on the right is a constraint in each sequent of the proof. Each logical rule simulates a transition rule of `lcc`, where commas on the left of sequents stand for parallel composition. For axioms, cut, contraction and $\wedge$-rules, it is evident. Idem for the left introductions of $\vee$ and $\top$. Since a constraint c contains only $\wedge$ and $\exists$, the only other right rule to consider is the right introduction of $\exists$, for which the induction hypothesis applies, by the definition of constraint entailment. For weakening, just remark that conjunction $\wedge$ is distributive with respect to $\vee$, and the hypothesis applies. The only non-trivial cases are:

1. the $\rightarrow$ left introduction:

$$\frac{\Gamma, A \vdash c \qquad \Delta \vdash d}{\Gamma, \Delta, d \rightarrow A \vdash c}$$

By induction hypothesis, $\Delta \longrightarrow (d_1 \wedge E_1) \vee \cdots \vee (d_k \wedge E_k)$, and for all j, $d_i \vdash_{\mathcal{C}} d$, so for all j, $d_j \wedge E_j \wedge (d \rightarrow A) \longrightarrow d_j \wedge E_j \wedge A$. Set $F = (d_1 \wedge E_1) \vee \cdots \vee (d_k \wedge E_k)$. Then $\Gamma \wedge \Delta \wedge (d \rightarrow A) \longrightarrow \Gamma \wedge F \wedge (d \rightarrow A) \longrightarrow \Gamma \wedge F \wedge A$. Again by induction hypothesis, $\Gamma \wedge A \longrightarrow (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$ and for all i, $c_i \vdash_{\mathcal{C}} c$. Set $G = (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$. Then $\Gamma \wedge \Delta \wedge d \rightarrow A \longrightarrow F \wedge G$, and the distributivity of $\wedge$ w.r.t. $\vee$ enables to conclude.

2. the $\exists$ left introduction:

$$\frac{\Gamma, A \vdash c}{\Gamma, \exists x A \vdash c}\ x \text{ not free in } \Gamma,\ c$$

By induction hypothesis, $\Gamma \wedge A \longrightarrow (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$, so $\exists x (\Gamma \wedge A) \longrightarrow \exists x ((c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n))$. Now x is not free in Γ, so $\exists x (\Gamma \wedge A) \equiv \Gamma \wedge \exists x A$, hence $\Gamma \wedge \exists x A \longrightarrow \exists x ((c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n))$, with $\exists x ((c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)) \vdash \exists x c$. Now x is not free in c, so $\exists x c \equiv c$, and the result is proved. ∎

3.2 Denotational semantics?

In this section we show through examples that the fixed points of (monotonic) cc agents (hence their denotational semantics [23, 10]) cannot be characterized via intuitionistic logic, even in the deterministic case.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a monotonic constraint system, and suppose, to simplify, that the set of declarations is empty. Again to simplify, we suppose that agents are deterministic. In [23], deterministic cc agents are viewed as continuous closure operators on the set of constraints $\mathcal{C}$; an agent A is determined by its set $[A]$ of fixed points:

$[c] = \{d \in \mathcal{C} \mid d \vdash_{\mathcal{C}} c\}$,
$[c \rightarrow A] = \{d \in \mathcal{C} \mid d \vdash_{\mathcal{C}} c \text{ implies } d \in [A]\}$,
$[A \wedge B] = [A] \cap [B]$,
$[\exists x A] = \{c \in \mathcal{C} \mid \text{ there exists } d \in [A] \text{ such that } \exists x c \dashv\vdash_{\mathcal{C}} \exists x d\}$.

By the first equality above, a logical characterization of the denotational semantics of cc agents should be: $c \in [A]$ iff $c \vdash_{IL(\mathcal{C},\mathcal{D})} A$. But then take $A = d \rightarrow B$: by the second equality, $c \vdash_{IL(\mathcal{C},\mathcal{D})} A$ iff $c \in [A]$ iff $(c \not\vdash_{\mathcal{C}} d$ or $c \in [B])$ iff $(c \not\vdash_{\mathcal{C}} d$ or $c \vdash_{IL(\mathcal{C},\mathcal{D})} B)$. This is impossible, since $c \vdash d \rightarrow B$ is by no means equivalent to $c \not\vdash d$ or $c \vdash B$: for instance, if c is $x > 3$, d is $x > 4$ and B is just the constraint $x > 5$, it is true that $c \not\vdash d$, but $c \not\vdash d \rightarrow B$. The problem is the instanciation of free variables.

Another approach to the denotational semantics of cc is the partial correctness criterion of [7].

3.3 Successes and suspensions?

The declarative nature of usual logic programming and constraint logic programming relies essentially on the logical observation of the successes (and failures) of a program. It is therefore natural to look for a similar characterization in the cc setting. We define the *successes* of a cc computation starting with A to be the stores c such that $A \longrightarrow c$. Note that, since $c \vdash_{\mathcal{C}} 1$ holds for any constraint c, a success may just be part of a final store. Other interesting observations would be the *suspensions* of a computation, i.e. the agents $B = c \wedge (d_1 \rightarrow A_1) \wedge \cdots \wedge (d_n \rightarrow A_n)$ such that $A \longrightarrow B$ and for no i, $c \vdash_{\mathcal{C}} d_i$.

Intuitionistic logic does not enable the observation of successes, neither that of suspensions, even in the deterministic case and without program declarations.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a constraint system.

▶ $\dashv$: It is not true in general that $A \dashv \Gamma$ (for Γ a success or a suspension) implies $A \longrightarrow \Gamma$. For instance $c \rightarrow d \dashv d$ but $c \rightarrow d$ can suspend, and have thus no success. Besides $d \dashv d \wedge (c \rightarrow d)$ and if we do not have $d \vdash c$, $d \wedge (c \rightarrow d)$ is a suspension, whereas the constraint c is not a suspension.

▶ $\vdash$: Similar problems arise with $\vdash$. $d \wedge (c \to A) \vdash d$ whereas $d \wedge (c \to A)$ suspends as soon as we do not have $d \vdash c$. Besides $d \wedge (d \to e) \vdash d \to e$, but $d \wedge (d \to e)$ has a success $(d \wedge e)$ and does not suspend.

▶ $\dashv\vdash$: Equivalence $\dashv\vdash$ is of no help. Suppose we do not have $d \vdash c$, and consider the following equivalence: $d \wedge (c \to d) \dashv\vdash d$. It does not allow to conclude anything about the operational behaviour of the agents d and $d \wedge (c \to d)$.

The obstacle is the structural rule of (left) *weakening*. Therefore we move to linear logic. At the same time it is natural to move to a non-monotonic version of `cc`, `lcc`, already introduced by Saraswat and Lincoln [22] and further studied by [4, 25].

4 Non-monotonic `cc`

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, and logical operators 1, the multiplicative conjunction $\otimes$, the existential quantifier $\exists$ and the exponential !; and $\Vdash_{\mathcal{C}} \subseteq \mathcal{C}^\star \times \mathcal{C}$. We assume $\otimes$ has neutral 1. Instead of $((c_1 \dots c_n), c) \in \Vdash_{\mathcal{C}}$, we write $c_1 \dots c_n \Vdash_{\mathcal{C}} c$.

$\vdash_{\mathcal{C}}$ is the least reflexive and transitive relation $\subseteq \mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the following rules:

$$\frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \qquad \frac{\Gamma \vdash c_1 \qquad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \qquad \frac{\Gamma, c \vdash d \qquad \Delta \vdash c}{\Gamma, \Delta \vdash d}$$

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

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

They are the rules of intuitionistic linear logic (ILL) for $\otimes$, $\exists$ and !, plus the cut rule. The syntax of `lcc` agents is given by the following grammar:

$$A ::= c \mid c \multimap A \mid A \otimes A \mid A \& A \mid \exists x A \mid p(\boldsymbol{x})$$

where $\otimes$ stands for parallel composition, and $\multimap$ for suspension. In an agent $A = c \otimes A_1 \otimes \cdots \otimes A_n$, if c is a constraint, the main constructor of each A_i is not $\otimes$, and no A_i is a constraint, we call c the *store* of A. It is the 'constraint' part of A.

Recursion is obtained with declarations:

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

• The *structural congruence* $\equiv$ is the least congruent equivalence such that $(\mathcal{A}/\!\equiv, \otimes, 1)$ is an abelian monoid, $(\mathcal{A}/\!\equiv, \&)$ is an abelian semi-group, and such that, for all agents A and B:

$$\exists x 1 \equiv 1 \qquad \exists x \exists y A \equiv \exists y \exists x A \qquad \frac{A \text{ and } B \text{ are } \alpha\text{-convertible}}{A \equiv B}$$

$$\frac{x \text{ is not free in } A}{\exists x(A \otimes B) \equiv A \otimes \exists x B} \qquad c \multimap (A \& B) \equiv (c \multimap A) \& (c \multimap B)$$

• The *transition* between agents $\longrightarrow$ is the least reflexive transitive congruence such that:

$$c \otimes (c \multimap A) \longrightarrow A \qquad \frac{c \vdash_{\mathcal{C}} d}{c \longrightarrow d} \qquad \frac{(p(\boldsymbol{x}) = A) \in P}{p(\boldsymbol{x}) \longrightarrow A}$$

$$\frac{A' \equiv A \qquad A \longrightarrow B \qquad B \equiv B'}{A' \longrightarrow B'}$$

$$A \& B \longrightarrow A \qquad A \& B \longrightarrow B \qquad A \otimes (B \& C) \longrightarrow (A \otimes B) \& (A \otimes C)$$

Remarks:

▶ Constraints are 'consumed' by suspensions; therefore the rule for $c \multimap A$ involves *non-determinism* since several stores may satisfy the condition of the rule.

▶ The non-deterministic $A \& B$ (already considered in [22, 4]) can behave either like A or like B, it has both capabilities. Note that this non-determinism is different from that of $\vee$ in monotonic `cc`.

Agents and declarations not involving & are said *deterministic*.

▶ The exponential ! allows to recover the monotonic case `cc`; for this reason, we just allow it on constraints.

Translation of deterministic `cc` into `lcc`

In the next section we show that IMALL (intuitionistic multiplicative additive linear logic) enables the observation of the successes of `lcc` agents. To be able to observe the successes of monotonic `cc` agents, one has to prove that `cc` agents can be translated into `lcc` ones, in such a way that the operational semantics is preserved. We do it for deterministic `cc` agents. The non-deterministic case can be translated into `lcc` as well, but it requires the use of the additive disjunction $\oplus$, which we do not consider in this paper to avoid confusion. We postpone the treatment of full monotonic `cc` to further work.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a constraint system. The linear constraint system $(\mathcal{C}, \Vdash_{\mathcal{C}})^\dagger$ has the same set of atomic constraints (with $\top$ renamed 1). The associated compound constraints and deterministic `cc` agents are translated into linear constraints and `lcc` agents:

$$\begin{array}{ll} c^\dagger = !c, \text{ if } c \text{ is atomic} & p(\boldsymbol{x})^\dagger = p(\boldsymbol{x}) \\ (c \rightarrow A)^\dagger = c^\dagger \multimap A^\dagger & (\exists x A)^\dagger = \exists x A^\dagger \\ \multicolumn{2}{c}{(A \wedge B)^\dagger = A^\dagger \otimes B^\dagger} \end{array}$$

Observe that c is a (monotonic) constraint iff $c^\dagger$ is a (linear) constraint. The proof of the following proposition is then straightforward:

Proposition 1 *Let c and d be monotonic constraints: $c \vdash_{cc} d$ iff $c^\dagger \vdash_{lcc} d^\dagger$. Let A and B be deterministic* `cc` *agents: $A \equiv_{cc} B$ iff $A^\dagger \equiv_{lcc} B^\dagger$, $A \longrightarrow_{cc} B$ iff $A^\dagger \longrightarrow_{lcc} B^\dagger$.*

For the atoms, $\wedge$ and $\exists$, our translation is Girard's second tranlation of intuitionistic logic into linear logic [8, p.81]. The only difference is the translation of $\rightarrow$ ($c^\dagger \multimap A^\dagger$ instead of $!(c^\dagger \multimap A^\dagger)$), which essentially forbids the erasure of suspensions.

5 Observing successes with IMALL

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

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a linear constraint system, and $\mathcal{D}$ a set of declarations. Let IMALL($\mathcal{C}$,$\mathcal{D}$) be the deduction system obtained by extending IMALL with:

- elements of $\Vdash_{\mathcal{C}}$ as non-logical axioms,
- for each declaration $p(\boldsymbol{x}) = A$ in $\mathcal{D}$, the sequent $p(\boldsymbol{x}) \vdash A$ as non-logical axiom.

We define the *successes* of an `lcc` computation starting with A to be the stores c such that $A \longrightarrow c$. The *suspensions* of A are the agents $B = c \otimes (d_1 \multimap A_1) \otimes \cdots \otimes (d_n \multimap A_n)$ such that $A \longrightarrow B$ and for no i, $c \vdash_{\mathcal{C}} d_i$.

Theorem 3 (Soundness) *Let A and B be* `lcc` *agents.*
If $A \equiv B$ then $A \dashv\vdash_{IMALL(\mathcal{C},\mathcal{D})} B$. If $A \longrightarrow B$ then $A \vdash_{IMALL(\mathcal{C},\mathcal{D})} B$.

Proof Trivial induction on $\equiv$ and $\longrightarrow$. ■

A *success* for an `lcc` agent A is a linear constraint c such that $A \longrightarrow c$.

Theorem 4 (Observation of successes) *Let A be an* `lcc` *agent, and c be a linear constraint. If $A \vdash_{IMALL(\mathcal{C},\mathcal{D})} c$, then $A \longrightarrow c$, i.e. c is a success for A.*

Sketch of proof It is simpler to prove the result for multisets of agents $A_1 \ldots A_n$: if $A_1 \ldots A_n \vdash c$ in IMALL($\mathcal{C}$,$\mathcal{D}$), then $A_1 \otimes \ldots \otimes A_n \longrightarrow c$. We proceed by induction on a (sequent calculus) proof of $A_1 \ldots A_n \vdash c$. Each logical rule simulates a transition rule of `lcc`, where commas on the left of sequents stand for parallel composition. For axioms, cut, $\otimes$-rules and !-rules, it is evident. Since a constraint c contains only $\otimes$, $\exists$ and $!$, the only other right rule to consider is the right introduction of $\exists$, for which the induction hypothesis applies, by the definition of constraint entailment. It is evident for the left introductions of & and 1:

$$\frac{\Gamma, A \vdash c}{\Gamma, A \& B \vdash c} \qquad \frac{\Gamma, B \vdash c}{\Gamma, A \& B \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

The only non-trivial cases are:

1. the $\multimap$ left introduction:

$$\frac{\Gamma, A \vdash c \qquad \Delta \vdash d}{\Gamma, \Delta, d \multimap A \vdash c}$$

By induction hypothesis, $\Delta \longrightarrow d$, so $(\Gamma \otimes \Delta \otimes d \multimap A) \longrightarrow (\Gamma \otimes d \otimes d \multimap A) \longrightarrow (\Gamma \otimes A)$. Again by induction hypothesis, $(\Gamma \otimes A) \longrightarrow c$, so $(\Gamma \otimes \Delta \otimes d \multimap A) \longrightarrow c$.

2. the $\exists$ left introduction:

$$\frac{\Gamma, A \vdash c}{\Gamma, \exists x A \vdash c}\; x \text{ not free in } \Gamma,\ c$$

By induction hypothesis, $(\Gamma \otimes A) \longrightarrow c$, so $\exists x(\Gamma \otimes A) \longrightarrow \exists x c$ But x is not free in c, so $\exists x c \equiv c$. And as x is not free in Γ, $\exists x(\Gamma \otimes A) \equiv (\Gamma \otimes \exists x A)$, hence $(\Gamma \otimes \exists x A) \longrightarrow c$. ■

Thanks to the translation from `cc` to `lcc`, the result holds for `cc` agents as well.

It is worth noting that a success is not proved to be a constraint entailing the initial agent (as in constraint logic programming), but entailed by the initial agent. This change of perspective is not very surprising in fact, since suspensions $c \multimap A$ contain implicitly a kind of negation (under the form of linear implication $\multimap$ as we shall see), which reverses the sense of deduction.

6 Discussion

We have shown that linear logic enables to do finer observations on `cc` and `lcc` agents than intuitionistic logic. This emphasizes the correspondence between the transition relation $\longrightarrow$ on agents and the entailment relation $\vdash$ in intuitionistic linear logic, and it is therefore a first step towards viewing concurrent constraint programming as (a fragment of some version of) linear logic. The purpose of the following paragraph is to make more precise the limits of this approach.

6.1 The limits of the correspondence

A look at the proof of Theorem 4 shows that the left introduction rules of IMALL sequent calculus just correspond to transitions in `lcc`. The rules for $\otimes$ express the monoidal structure of $\otimes$. Program declarations have been oriented $(p(\boldsymbol{x}) = A$

is translated into $p(\boldsymbol{x}) \vdash A$): then an axiom $p(\boldsymbol{x}) \vdash A$ just corresponds to the replacement of the agent $p(\boldsymbol{x})$ by the agent A.

On the contrary, the right introductions of &, $\multimap$ and $\exists$ do not correspond to transitions of agents. Let us look at them more closely:

- The right introduction of &:

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

  expresses the *combination of two experiments* starting with the same agent. You know that $A\&B$ can behave either like A or like B (left introduction), and now you postulate the converse, i.e. that observing either A or B from Γ means $\Gamma \longrightarrow A\&B$. The right introduction of & is equivalent to $A\&A \dashv\vdash A$.
- The right introduction of $\multimap$:

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

  expresses a '*porosity*' of suspensions: for instance, $A \otimes (c \multimap B) \vdash c \multimap (A \otimes B)$... A strong form of porosity is considered in process calculi like the π-calculus [16], namely *enablement*: for a guard ω such that no free variable in A is bound in ω, it states that $\omega(A \otimes B) \equiv A \otimes (\omega B)$. But it is not considered in `cc`, and as we shall see in the next paragraph, it forbids the observation of suspensions.
- The right introduction of $\exists$:

$$\frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}$$

  expresses *blindness*. The agent refuses the communication with the environment through variable (or channel) x, what can lead it to deadlock.

The significance of these remarks is twofold:

1. they are limits to the correspondence between deduction in IMALL and the operational semantics of `cc` and `lcc` languages,
2. their rather intuitive operational interpretation (at least for the first two ones) deserves further consideration.

6.2 Suspensions

Concurrent constraint programming suggests another interesting observation, namely the suspensions of an agent: a *suspension* for an `lcc` agent A is an agent $B = c \otimes (d_1 \multimap A_1) \otimes \cdots \otimes (d_n \multimap A_n)$ such that $A \longrightarrow B$ and for no i, $c \vdash_{\mathcal{C}} d_i$.

The above remark on the right introduction of $\multimap$ shows that suspensions cannot be observed in the setting of IMALL: for instance, $c \otimes (c \multimap 1) \vdash c \multimap$

($c\otimes 1$), a suspension, whereas $c\otimes(c \multimap 1)$ succeeds with 1. This 'porosity' is linked to the lack of a *sequential composition* connective in ILL. Such a connective needs to be non-commutative. Abrusci [1] studied a pure non-commutative version LL, without commutative and additive (&) connectives, and we need at least a commutative multiplicative connective (the 'parallel' connective). Retoré's *before* connective $<$ [20] is not a solution either, since $A \otimes (B < C) \vdash B < (A \otimes C)$.

The next step of our investigation will be to define a non-commutative version of linear logic, which copes with this difficulty.

6.3 The significance for linear logic and concurrency

Other (linear) logical approaches have been proposed to study concurrency with the approach of logic programming. Andreoli and Pareschi [2] point out that the 'proof-search as computation' analogy for linear logic corresponds to a reactive paradigm, but in Linear Objects, synchronous message passing involves extra-logical operators ('tell markers'), whereas concurrent constraint programming is asynchronous, what avoids the resort to such extra-logical operators. Miller [15] describes a connection between the π-calculus and linear logic, but uses non-logical constants as well; a connection *à la* Miller between Boudol's asynchronous version of the π-calculus [5] and our work should be interesting. Perrier [18] proposes a denotational semantics based on the phase semantics, to model the interaction capability of a process. Our paper focuses on concurrent constraint programming, so the two approaches are different, but we think a thorough comparison would be interesting. Kobayashi and Yonezawa [11] define several concurrent semantics for linear logic processes, including bisimulation; we believe the relationship with concurrent constraint programming through our work should be worth investigating.

Acknowledgements

I am greatly indebted to François Fages for many useful discussions on this work. I thank the anonymous referees for their comments on a preliminary version of this paper.

References

1. M. Abrusci. Phase semantics and sequent calculus for pure non-commutative classical linear logic. *Journal of Symbolic Logic*, 56(4), 1991.
2. J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, 9, 1991.
3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.
4. E. Best, F.S. de Boer, and C. Palamidessi. Concurrent constraint programming with information retrieval. Esprit project ACCLAIM final report, 1994.
5. G. Boudol. Asynchrony and the π-calculus. Technical Report RR 1702, INRIA, 1992.

6. F. Fages. Constructive negation by pruning. *To appear in J. of Logic Programming*, 1996.
7. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. In *Proceedings of the ACM International Conference on Principles of Programming Languages*, 1994.
8. J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.
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. R. Jagadeesan, V. Shanbhogue, and V.A. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Parc, 1991.
11. 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.
12. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(3):289–308, 1987.
13. P. Lincoln and V.A. Saraswat. Proofs as concurrent processes. Manuscript, 1992.
14. M.J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of ICLP'87, International Conference on Logic Programming*, 1987.
15. D. Miller. The π-calculus as a theory in linear logic: preliminary results. In *Proceedings Workshop on Extensions of Logic Programming*, Springer LNCS 660, 1992.
16. R. Milner. The polyadic π-calculus. Technical Report ECS-LFCS-91-180, Edinburgh University, 1991.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1), 1992.
18. G. Perrier. Concurrent programming in linear logic. Technical Report CRIN 95-R-052, INRIA-Lorraine, 1995.
19. 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.
20. Ch. Retoré. *Réseaux et séquents ordonnés*. PhD thesis, Université Paris 7, 1993.
21. V.A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.
22. V.A. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming. Manuscript, 1992.
23. 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.
24. P.J. Stuckey. Constructive negation for constraint logic programming. *Information and Computation*, 118(1), 1995.
25. C. Tse. The design and implementation of an actor language based on linear logic. Master Thesis, MIT, 1994.

This article was processed using the LaTeX macro package with LLNCS style