Phase Model Checking for some Linear Logic Calculi

Sylvain Soliman
Sylvain.Soliman@inria.fr

Abstract

Building upon previous work in the logical semantics of linear concurrent constraint programming languages (LCC), an example of linear logic based calculus, we design an original phase model checking method for proving safety properties of programs. We describe our implementation of the phase model checker using constraint programming techniques and provide first experimental results.

1 Introduction

The class of Concurrent Constraint programming languages (CC) was introduced a decade ago by Vijay Saraswat as a unifying framework for constraint logic programming and concurrent logic programming [11]. The CC paradigm constitutes a representative abstraction of constraint programming languages that are used nowadays in a wide variety of application domains, most notably for solving combinatorial search problems.

One can give a simple translation of CC programs in Jean-Yves Girard's Linear Logic [6] that is sound and complete for observing the entailment-closed sets of successes and accessible stores [5, 4, 12]. This, among other reasons, led to a generalization of CC languages, called Linear Concurrent Constraint languages (LCC), that is used throughout the paper for the presentation of our results. The class LCC is obtained from CC simply by allowing constraint systems based on linear logic instead of classical logic. This is a generalization which greatly extends the expressive power of CC, as non monotonic evolutions of the store are possible through the consumption of constraints by ask agents [3, 12] but it still enjoys a simple linear logic semantics with the same soundness and completeness properties. These results have been used in [4, 5] to prove safety properties of LCC programs simply by exhibiting special phase models of programs.

This paper presents an application of these results to the proof of safety properties of LCC programs, through the implementation of a model checker based on the phase semantics of linear logic, as first proposed in [4]. We provide first experimental results on the efficiency of the phase model checker for proving the safety of some simple LCC programs for specifying protocols. Finally we conclude on the perspectives of this work and the generalizations of this procedure to other calculi.

2 Singleton Based Phase Model Checking

It is natural to use the logical characterization of LCC observables (see appendix A) to prove properties of programs. In [4] we used the phase semantics of Linear Logic, as given in [6, 9] to devise a new method that revealed useful especially for proving safety properties of LCC programs. We now describe how it can be implemented with some restrictions using efficient constraint programming techniques.

Recall that, to prove a property such as A never reaches a store containing c, the soundness of the linear logic semantics allowed us to look for a proof that $A \nvdash c \otimes \top$. Any calculus where properties can be expressed like that can then rely on the soundness of the phase semantics to ensure that it is enough to exhibit a phase space and an interpretation of formulae (agents and constraints) into facts, compatible with the non-logical axioms, and such that $[\![A]\!] \not\subset [\![c]\!]$ (see appendix B).

Our approach to finding the phase space and the valuation is now based on a few remarks:

- We want to find everything *statically*, so we will not use any of the methods like those of Okada and Terui [10], which generate the phase space from a proof search, thus an execution.

- We want to enumerate some spaces, but we cannot try them all. We thus have to restrict our search to some *special* kinds of phase spaces, where we will be able to reason easily (to prove the non-inclusion).
- In [14] Yilma tried this kind of approach, restricting himself to *small* and *cyclic* structures. But then he had to enumerate all the possible valuations to find the counter-example for the inclusion.
- Almost all the examples that we treated (in [4] and later) have solutions in very simple phase structures, where all sets are facts and the valuation associates only singletons to formulae.

These remarks, and especially the last one, which transforms inclusion conditions to equalities, led us to use the constraint logic programming tools as a basis for our *phase model checker*.

2.1 Implementation

Our implementation of the phase model checker uses $\mathbb{N}$ as the basic monoid for all our phase structures, not only because it is what was used in the examples of [4], but also because it builds, in some sense, the free monoid based on the conditions of the non-logical axioms. The prime numbers will thus be a good basis for our interpretation.

The system then searches for a valuation, i.e. it associates an integer to each atomic formula, such that the equalities coming from the non-logical axioms are respected. In other words, these conditions are equality constraints on the variables associated with each atomic formula. The non-inclusion being the only dis-equality constraint.

An implementation of these ideas was realized, using the constraint logic programming language GNU-Prolog with constraint solvers over finite domains.

2.2 Examples

Here is a non-recursive version of the dining philosophers program in LCC.

```
goal :- phil1,phil2,phil3,phil4,phil5,fork1,fork2,fork3,fork4,fork5.
phil1 :- fork1,fork2 -> eat1,(eat1 -> fork1,fork2,phil1).
phil2 :- fork2,fork3 -> eat2,(eat2 -> fork2,fork3,phil2).
...
phil5 :- fork5,fork1 -> eat5,(eat5 -> fork5,fork1,phil5).
```

We note `','` for $\otimes$ and $\|$, depending on position, and `'->'` for $\rightarrow$.

Here is the result of a simple execution, checking that philosophers 1 and 2 can never eat at the same time:

```
| ?- find_phase('philo_5',[goal],[([eat1,eat2],top)]).
eat5=1 eat3=1 eat1=2  fork4=1 fork2=2 phil5=1 phil3=1 phil1=1
eat4=1 eat2=2 fork5=1 fork3=1 fork1=1 phil4=1 phil2=1 goal=2
```

Actually here is a more detailed explanation of what happens: 16 variables corresponding to atomic constraints (5 for each `phil`, `fork` and `eat` plus one for `goal`) are declared; 10 other variables corresponding to *asks* are declared; 16 equality constraints are given to the solver: one for each *ask* and one for each declaration; finally the dis-equality constraint corresponding to the safety property is added.

If we had n philosophers, we would thus have $5n+1$ variables, with $3n+1$ equality constraints and one dis-equality. You can see at the end of this paper results for bigger instances of this problem.

Note that a simple constraint such as `A #\= B` would only allow to check properties as $A \nvdash B$, however we often want to check if $A \nvdash B \otimes \top$. Therefore we have to rely on integer arithmetic to notice that if m, n and p are integers $\forall p, m \neq n \cdot p$ is equivalent to m *modulo* $n \neq 0$, and GNU-Prolog's FD solver nows how to handle these as `A rem B #\= 0`.

3 Going farther

3.1 Coping with incompleteness

The method described here is however not complete for different reasons:

One is that if we are based on finite domains, we will never be able to handle properly programs that have an unbounded parameter. In some cases we will be able to collapse all the values after some bound, but for instance if you take the dining philosophers with `N` not given, it is very hard to find a phase space as that given in [4], as it relies on an infinite number of primes.

Another reason is linked with the choice to consider only singletons. This has the important drawback of creating confusions. For instance with the following program: $\mathtt{P = tell(d)}, \mathtt{Q = c} \rightarrow \mathtt{P}$, a singleton-based phase structure does not allow one to prove that `c` is not accessible from `P`, i.e. $\mathtt{P} \not\vdash \mathtt{c} \otimes \top$, as we can deduce $[\![\mathtt{P}]\!] = [\![\mathtt{c}]\!] \cdot [\![\mathtt{Q}]\!]$ from the second declaration.

Finally, all integers are invertible, that is $m \cdot p = n \cdot p$ implies $m = n$. Thus we cannot capture the idea that in the first case the *presence* of p is necessary to go from m to n. Therefore we cannot prove much for programs such as Peterson's mutual exclusion algorithm given below, as there is a *check* on the value of a variable that may not change its value but allows a process to go from `off` to `on` (see below).

```
p1 :- off1,turn1 -> on1,turn2,wait1.
p1 :- off1,turn2 -> on1,turn2,wait1.
wait1 :- off2 -> off2,cs1,(cs1,on1 -> off1,p1).
wait1 :- turn1 -> turn1,cs1,(cs1,on1 -> off1,p1).
(same for p2 and wait2)
init :- off1,off2,turn1,p1,p2.
```

It is important to note however that one can cope with all these problems to show some important properties, by simply using a more precise way of expressing what is to be shown. Let us consider again the previous example of Peterson's mutual exclusion algorithm. It is shown in [14] that even if it is not provable directly that $\mathtt{init} \not\longrightarrow^* \mathtt{cs1} \otimes \mathtt{cs2} \otimes \top$, one can use the phase semantics to prove it by remarking that the only four ways to reach such a state are states of the following forms:$\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{turn1} \otimes \mathtt{turn2} \otimes \top$, $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{turn1} \otimes \mathtt{off1} \otimes \top$, $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{off2} \otimes \mathtt{turn2} \otimes \top$ and $\mathtt{on1} \otimes \mathtt{on2} \otimes \mathtt{off1} \otimes \mathtt{off2} \otimes \top$.

The phase model checker can then compute:

```
| ?- find_phase('peterson',[init],[([on1,on2,turn1,turn2],top),
| ?- ([on1,on2,turn1,off1],top),([on1,on2,off2,turn2],top),
| ?- ([on1,on2,off1,off2],top)]).
init=8 wait2=1 p2=1  off2=2  turn2=2 off1=2  p1=1
cs2=1  on2=2   cs1=1 wait1=1 on1=2   turn1=2
```

3.2 Doubletons and finite sets

We have seen above that almost all the limits of our method can be coped with, however the singleton restriction might sometimes be too restrictive, for instance in the very simple producer/consumer protocol given below:

```
p :- dem -> pro,p.                    c :- pro -> dem,c.
init :- dem,dem,dem,p,p,p,p,p,c,c.
```

One can check that it is impossible to prove with singletons that a producer cannot consume, i.e. $p \parallel pro \not\longrightarrow dem \parallel \Gamma$, as the definition of p gives $p \cdot pro = p \cdot dem$.

Solvers on sets would allow us to solve this problem, it is however noticeable that a simple modification of our solver allows it to handle *doubletons* by simply using GNU-Prolog's list constructor, and that we thus already have a more general solver.

```
| ?- find_dbl('prod_cons',[p,pro],[([dem],top)]).
p=[1,1] pro=[2,1] dem=[2,2] c=[0,0]
```

Such a modification should allow us to use bounded lists of a size much bigger than two, however our solver gets really slow and the lack of a specialized solver is severely felt.

4 Experimental results

The following table sums up our experimental results. The size of the example indicates the number of initial agents. The numbers of variables and constraints indicate the size of the constraint satisfaction problem issued from the phase model checker.

protocol	size	number of variables	number of constraints	time for proving safety
prod./cons. (doubletons)	2	12	5	3850 ms
dining philosophers	5	26	16	480 ms
dining philosophers	10	51	31	3470 ms
dining philosophers	20	101	61	44730 ms
Peterson's algorithm	2	25	25	1420 ms

5 Conclusion

To be efficient, program verification needs abstractions in order to get rid of useless execution details. The semantics of concurrent constraint programs in linear logic and in its model theory of provability, namely phase semantics, provides an abstract interpretation of LCC programs which is given by the logic of LCC agents that is built-in inside a phase model checker that we have implemented with some restrictions.

We have shown that these restrictions, while making it possible to use the constraint programming technology to search for phase models with arithmetic constraints, still allow us to find automatically some non-trivial proofs of safety properties of LCC programs. The efficiency issue will of course need more work than our first prototype implementation, but the technology already available in constraint solvers gives us great hope for improvement.

Finally, as the only part really relevant to LCC in this study was the expressibility, through its semantics, of safety properties in linear logic, we believe that our method can thus be used for other calculi based on linear logic and hope to extend concretely the prover in that direction in the future. The first extension is that of handling the family of logic programming languages based on linear logic, like LO [1], Lolli [8], or Lygon [7], where only a new parser is needed (the properties are already expressed as linear logic formulae). Other coordination languages or rule based languages may also have a sound linear logic semantics, so there is still a lot of experimentation to do.

References

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

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

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

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

[5] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 164, 2001. Available from http://pauillac.inria.fr/~fages/Papers/FRS01ic.ps.

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

[7] J. Harland, D. Pym, and M. Winikoff. Programming in lygon: An overview. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, Munich*, pages 391–405, July 1996.

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

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

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

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

[13] S. Soliman. *Programmation concurrente avec contraintes et logique linéaire.* PhD thesis, Université Paris 7, Denis Diderot, 2001.

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

A Appendix: LCC

A.1 Syntax and operational semantics

In this paper, a set of variables is denoted by X, Y,..., the set of free variables occurring in a formula A is denoted by $\mathrm{fv}(A)$, a sequence of variables is denoted by $\vec{x}$, $A[\vec{t}/\vec{x}]$ denotes the formula A in which the free occurrences of variables $\vec{x}$ have been replaced by terms $\vec{t}$ (with the usual renaming of bound variables for avoiding variable clashes).

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

The essential difference between LCC and CC is that constraints are formulae of linear logic and that communication (the *ask* rule) consumes information.

Definition A.1 (Linear constraint system) *A* linear constraint system *is a pair* $(\mathcal{C}, \vdash_\mathcal{C})$*, where:*

- $\mathcal{C}$ *is a set of formulae (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, and the existential quantifier $\exists$;*
- $\Vdash_\mathcal{C}$ *is a subset of $\mathcal{C} \times \mathcal{C}$ which defines the non-logical axioms of the constraint system.*
- $\vdash_\mathcal{C}$ *is the least subset of $\mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_\mathcal{C}$ and closed by the following rules of intuitionistic linear logic:*

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

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

The syntax of LCC *agents* is given in table 1, where $\|$ stands for parallel composition, $+$ for non-deterministic choice, $\exists$ for variable hiding and $\rightarrow$ for blocking ask. The atomic agents $p(\vec{x})\ldots$ are called *process calls* or *procedure calls*, we assume that the arguments in the sequence $\vec{x}$ are all distinct variables. The *ask* agent in LCC is written with a universal quantifier in order to make explicit the variables which are bound in the guard.

Recursion is obtained by declarations. We make the usual hypothesis that in a declaration $p(\vec{x}) = A$, all the free variables occurring in A occur in $\vec{x}$. The set of declarations of an LCC program, denoted by

Agents	$A ::= p(\vec{x}) \mid tell(c) \mid (A \parallel A) \mid A + A \mid \exists x A \mid \forall \vec{x}(c \rightarrow A)$
Declarations	$D ::= \epsilon \mid p(\vec{x}) = A \mid D, D$
Program	$P ::= D.A$
α-Conversion	$\dfrac{z \notin fv(A)}{\exists y A \equiv \exists z A[z/y]}$
Parallel comp.	$A \parallel B \equiv B \parallel A$ $A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$
Equivalence	$\dfrac{(X; c; \Gamma) \equiv (X'; c'; \Gamma') \longrightarrow (Y'; d'; \Delta') \equiv (Y; d; \Delta)}{(X; c; \Gamma) \longrightarrow (Y; d; \Delta)}$
Tell	$(X; c; tell(d), \Gamma) \longrightarrow (X; c \otimes d; \Gamma)$
Ask	$\dfrac{c \vdash_{\mathcal{C}} d[\vec{t}/\vec{y}] \otimes e}{(X; c; \forall \vec{y}(d \rightarrow A), \Gamma) \longrightarrow (X; e; A[\vec{t}/\vec{y}], \Gamma)}$
Hiding	$\dfrac{y \notin X \cup fv(c, \Gamma)}{(X; c; \exists y A, \Gamma) \longrightarrow (X \cup \{y\}; c; A, \Gamma)}$
Procedure calls	$\dfrac{(p(\vec{y}) = A) \in D}{(X; c; p(\vec{y}), \Gamma) \longrightarrow (X; c; A, \Gamma)}$
Blind choice	$(X; c; A + B, \Gamma) \longrightarrow (X; c; A, \Gamma)$ $(X; c; A + B, \Gamma) \longrightarrow (X; c; B, \Gamma)$

Table 1: LCC syntax and operational semantics.

$\mathcal{D}$, is the closure by variable renaming of a set of declarations given for distinct procedure names p. A *program* $D.A$ is a declaration D together with an initial agent A.

The operational semantics is defined on configurations where the store is distinguished from agents. A *configuration* is a triple $(X; c; A)$, where c is a constraint called the store, A is an agent or $\emptyset$ if empty, and X is a set of variables, called the *hidden* variables of c and A. The operational semantics is defined in the style of the CHAM [2] by a transition system which does not take into account specific evaluation strategies. The *structural congruence* $\equiv$ is the least congruence satisfying the rules of table 1. For convenience here, and unlike in [5], the logical equivalence of constraints is not built-in in the congruence. We write $\Gamma, \Delta, ...$ for multisets of agents in configurations. Congruence is extended to multisets of agents in the obvious way: $\Gamma \equiv \Gamma'$ iff $\Gamma = \{A_1, \ldots, A_n\}$, $\Gamma' = \{A'_1, \ldots, A'_n\}$ and $\forall i = 1, \ldots, n$, $A_i \equiv A'_i$. Two configurations are said *congruent*, $(X; c; \Gamma) \equiv (X'; c'; \Gamma')$, when the sets X and X' are equal, the constraints c and c' are $\mathcal{C}$-equivalent, and the multisets of agents Γ and Γ' are congruent. The *transition* relation $\longrightarrow$ is the least transitive relation on configurations satisfying the rules of table 1.

A.2 LCC logical semantics

LCC programs have a simple semantics in linear logic. In this section we give translations of LCC agents by logical formulae which are sound and complete for the observations of success stores, accessible stores and terminal stores.

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

Definition A.2 *LCC agents are translated into intuitionistic linear logic formulas in the following way:*

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

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

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

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

Theorem A.3 (Soundness) *Let $(X;c;\Gamma)$ and $(Y;d;\Delta)$ be CC configurations.*
If $(X;c;\Gamma) \equiv (Y;d;\Delta)$ then $(X;c;\Gamma)^\dagger \dashv\vdash_{ILL(\mathcal{C},\mathcal{D})} (Y;d;\Delta)^\dagger$.
If $(X;c;\Gamma) \longrightarrow^* (Y;d;\Delta)$ then $(X;c;\Gamma)^\dagger \vdash_{ILL(\mathcal{C},\mathcal{D})} (Y;d;\Delta)^\dagger$.

The reader can refer to [5, 13] for the results concerning completeness with respect to different observables (stores, successes, suspensions, etc...) and for the proofs of all the theorems.

B Appendix: Phase Semantics

B.1 Phase semantics of intuitionistic linear logic

Phase semantics is the natural provability semantics of linear logic [6]. 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 A). Nevertheless it is simpler to recall Okada's definition of the phase semantics for full intuitionistic LL [9] and to extend it to constants ($\mathbf{1}$, $\mathbf{0}$, $\top$).

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

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

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

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

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

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

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

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

Definition B.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 B.3 *Given an enriched phase space, a* valuation *is a mapping* η *from atomic formulas to facts such that* $\eta(\top) = \top$*,* $\eta(\mathbf{1}) = \mathbf{1}$ *and* $\eta(\mathbf{0}) = \mathbf{0}$*.*

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

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

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

Definition B.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 B.5 (Soundness [6, 9]) *If there is a sequent calculus proof of* $\Gamma \vdash A$ *then* $\models (\Gamma \vdash A)$*.*

Theorem B.6 (Completeness [6, 9]) *If* $\models (\Gamma \vdash A)$ *then there is a sequent calculus proof of* $\Gamma \vdash A$*.*

B.2 Proving safety properties of LCC programs with the phase semantics

Using the phase semantics presented above we can prove safety properties of LCC programs. The theorem B.5 of soundness of the phase semantics w.r.t. ILL can easily be extended to $ILL_{\mathcal{C},\mathcal{D}}$ by imposing to any valuation η to satisfy the inclusions coming from the non-logical axioms (the axiom $c \vdash d$ imposes $\eta(c) \subset \eta(d)$).

By contrapositive we get:

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

which is equivalent to:

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

As the contrapositive of the theorem A.3 of soundness from LCC to $ILL_{\mathcal{C},\mathcal{D}}$ is:

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

We have:

Proposition B.7 *To prove a safety property of the kind:* $(X;c;\Gamma) \not\longrightarrow (Y;d;\Delta)$, *it is enough to show that:*
$\exists$ *a phase space* $\mathbf{P}$, *a valuation* η, *and an element* $a \in \eta((X;c;\Gamma)^\dagger)$ *such that* $a \notin \eta((Y;d;\Delta)^\dagger)$.

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