Concurrent constraint programming and linear logic

Sylvain Soliman

UNIVERSITE PARIS 7 - DENIS DIDEROT
UFR INFORMATIQUE

Année 2000-2001

N° attribué par la bibliothèque

THESE

pour l'obtention du Diplôme de

DOCTEUR DE L'UNIVERSITE PARIS 7
SPECIALITE: PROGRAMMATION

présentée et soutenue publiquement
par

Sylvain SOLIMAN

le 20 Avril 2001

Titre :

Programmation Concurrente avec Contraintes et Logique Linéaire

Directeur de thèse :

François Fages

JURY

Mme	Catuscia Palamidessi	Rapporteur
M.	Philippe de Groote	Rapporteur
M.	Vincent Danos	
M.	Paul Ruet	
M.	François Fages	

"One of the symptoms of an approaching nervous breakdown
is the belief that one's work is terribly important."
– Bertrand Russel

Remerciements.

Je tiens tout d'abord à remercier ceux qui m'ont aidé à écrire cette thèse, et en particulier François Fages, qui a accepté d'encadrer mon travail, d'abord au sein du Laboratoire d'Informatique de l'École Normale Supérieure, puis à l'INRIA. Ses conseils avisés furent nombreux et précieux, ils m'ont guidé dans mes recherches, aussi bien en Informatique et en Logique qu'en ce qui concerne les Vélos Tout-Terrain. Sa disponibilité à toute épreuve, même lorsque la "liste noire" des tâches à finir pour la semaine précédente s'allongeait, me fut d'un grand secours pour orienter mon travail, et son enthousiasme de tous les jours fut sans le moindre doute une des raisons qui m'ont permis de ne pas laisser cette thèse inachevée, a l'instar de certains romans...

Je remercie également mes rapporteurs, Catuscia Palamidessi et Philippe de Groote, c'est grâce à eux que mon manuscrit à pu évoluer vers sa forme définitive, ainsi que tous ceux qui ont fait l'effort de lire cette thèse et en particulier les membres du jury: Vincent Danos et Paul Ruet, mais aussi Dale Miller, Rajeev Gore, Victor Vianu, et Mitsuhiro Okada pour les nombreuses discussions que nous avons eu pendant les trois dernières années.

J'adresse par la même occasion mes plus plates excuses à tous ceux qui ont du me subir au jour le jour pendant ces années de thèse, les membres de l'équipe Langages Logiques et Contraintes du LIENS, et en particulier Paul Ruet qui fut mon voisin de bureau, Gabriele Santini dont j'avoue avoir squatté le bureau plus que ce qu'aurait justifié nos postes de *responsable de la machine à café*, mais aussi les membres du projet Contraintes de l'INRIA et en particulier Sorin Craciunescu et Emmanuel Coquery à qui je reconnais avoir imposé des pauses intempestives et des appels au déjeuner quelque peu brutaux, et enfin ceux du Club de Go de l'INRIA qui j'espère survivra à mon départ.

Je remercie mes camarades non-informaticiens qui pendant trois ans ont fait admirablement semblant de s'intéresser à l'avancement de ma thèse, malgré mon invariable réponse "ça suit son cours...", j'espère que le titre de mon stage de DEA les fera sourire encore longtemps.

Enfin je remercie mon épouse, Yuki, pour une pléthore de raisons dont la présence à mes côtés n'est pas la moindre, ainsi que mes parents sans qui je ne serais évidemment pas ici.

"Why waste time learning, when ignorance is instantaneous?"
– Calvin and Hobbes

Résumé

Dans cette thèse, nous étudions les liens étroits entre la logique linéaire et la programmation concurrente par contraintes, sous l'angle de la sémantique, et plus précisément de la vérification de programmes.

Nous raffinons les observables caractérisables grâce à la logique linéaire et étendons des résultats antérieurs, afin d'obtenir une sémantique plus précise et plus générale. Ces résultats sont obtenus par une traduction plus fidèle des agents en formules logiques et un enrichissement de la théorie.

Nous présentons aussi une méthode originale de preuves de programmes, basée sur la sémantique de la prouvabilité de la logique linéaire: la sémantique des phases. Celle-ci nous donne un outil de vérification avec de nombreuses bonnes propriétés: la possibilité de montrer une propriété universelle du programme simplement par un contre-exemple; une facilité d'abstraction; enfin la simplicité des preuves obtenues.

Nous construisons donc une méthode systématique de preuve et, après l'avoir testée avec succès sur des exemples classiques, nous étudions son automatisation. Cette étude aboutit à l'implémentation d'un prototype de prouveur qui accompagne un interpréteur pour les langages linéaires concurrents avec contraintes.

Abstract

In this thesis, we study the close links between linear logic and concurrent constraint programming, from the angle of semantics and more precisely of program verification.

We refine the observables that can be characterized in linear logic and extend existing results to obtain a more precise and more general semantics. These results are only based on a more faithful translation from agents into logic formulae and on an enrichment of the theory.

We also present an original method to get program proofs, based on the provability semantics of linear logic: phase semantics. This gives us a tool for verification, enjoying lots of good properties, like the ability to prove a universal property of the program simply with a counter-example; an easy abstraction from the program; and the simplicity of the proofs obtained.

We thus construct a systematic method of proof generation and after applying it to classical examples, we study its automatization. This research leads to the implementation of a prototype-prover which can live side by side with our interpreter for linear concurrent constraint languages.

Table des matières

Chapitre 1

Introduction

"Understanding is a kind of ecstasy."
– Carl Sagan

Langages concurrents avec contraintes

La *Programmation Logique avec Contraintes* (CLP), introduite par Jaffar et Lassez en 1987 [16] est une extension de la programmation logique basée sur la combinaison avec les clauses de Horn définissant les relations, d'un langage de premier ordre – les contraintes – décidable, sur une structure donnée $\mathcal{X}$ représentant le domaine du discours. Ce paradigme très général a été, depuis lors, utilisé avec succès dans de nombreuses applications de modélisation et d'optimisation [17]. De plus, la sémantique logique des programmes logiques s'étend naturellement aux langages CLP($\mathcal{X}$) pour caractériser les succès et les échecs finis, il suffit en effet de considérer les conséquences en logique classique, non seulement du programme, mais aussi de la théorie de la structure $\mathcal{X}$.

Les langages *Concurrents avec Contraintes* (CC($\mathcal{X}$)), introduits la même année par Maher [24] puis Saraswat [33], peuvent être vus comme une extension de la classe CLP($\mathcal{X}$) par un mécanisme de synchronisation basé sur l'implication de contraintes. Le modèle obtenu permet à des agents de communiquer au travers d'une sorte de "tableau noir", le *store*, qui est en fait une contrainte exprimant une accumulation d'information partielle sur les variables du calcul. Les opérations de base des agents sont donc d'apporter l'information c au store par l'opérateur *tell(c)* ou d'attendre que le store implique une information donnée par l'opérateur *ask(c)*. En ajoutant le non-déterminisme, on obtient des langages assez expressifs pour permettre, entre

autres, une reconstruction de certains processus utilisés en pratique dans les implémentations de CLP mais "cachés" jusqu'alors, comme par exemple le prédicat `freeze` de Prolog [9, 6], la propagation de contraintes dans les domaines finis [15], etc...

Sémantique

L'évolution monotone du store au cours d'un calcul CC a permis de doter les langages CC($\mathcal{X}$) d'une sémantique dénotationnelle simple, basée sur une représentation des agents par des opérateurs de fermeture [35, 18]. Cependant, même si la grande idée *Exécution = Recherche de preuve* de la programmation logique est moins présente dans les langages CC($\mathcal{X}$), l'équation *Programme + Domaine = Théorie* reste valable et a donné lieu à une sémantique logique des CC complémentaire de la sémantique dénotationnelle.

Or, de la grande variété des applications de CC vient naturellement une volonté de pouvoir démontrer certaines propriétés des programmes, par exemple des propriétés de sûreté ou de terminaison. Mais si ce domaine a été très étudié dans le cadre des langages purement concurrents (model-checking, temporal logic, ...), peu de choses ont été faites pour les CC. Certains travaux ont essayé de tirer parti de la sémantique dénotationnelle [10], nous suivons ici une voie différente qui se fonde sur l'idée qu'en utilisant la Logique Linéaire (LL) introduite par Girard [14] en 1987 (un bon millésime visiblement) on peut tirer un meilleur parti de l'équation *Programme + Domaine = Théorie*.

Cette idée a permis, non seulement d'affiner les résultats de complétude de la sémantique logique pour CC [30], ce qui est important pour les preuves de propriétés de vivacité (voir chapitre 3), mais a aussi conduit à l'enrichissement du langage lui-même: on est en effet naturellement amené à considérer des domaines de contraintes *linéaires* et à supposer que les opérations d'ajout et d'implication de contrainte (*ask* et *tell*) correspondent aux conjonctions et implications de LL. Le store peut alors éventuellement évoluer de manière non-monotone par "consommation" d'une contrainte. L'expressivité de cette nouvelle classe de langages Linéaires Concurrents avec Contraintes (LCC($\mathcal{X}$)) [12], très proche des ccp_{get} de Best et al. [4], nous permet enfin d'écrire de façon déclarative les protocoles concurrents que CC mettait à notre portée mais gérait maladroitement (par exemple par des streams, ...).

Plan de la thèse

Dans le chapitre 2 nous établissons les notations qui seront utilisées dans cette thèse et rappelons les définitions classiques pour CC et LCC, nous les agrémentons d'exemples et discutons le problème de la séquentialité dans CC, puis le chapitre 3 donne les résultats de base concernant la sémantique logique des langages concurrents avec contraintes. L'accent est mis sur les théorèmes de complétude et leur limitations, ainsi que sur une comparaison avec les résultats obtenus par l'approche sémantique dénotationnelle.

Nous montrons ensuite dans le chapitre 4 comment ces résultats ont pu être poussés plus loin à l'aide d'une sémantique logique plus "fine", aboutissant ainsi a une sémantique précise de type *input-output*, comme seule la sémantique dénotationnelle le permettait jusqu'alors, mais en s'affranchissant des limitations sur le langage, comme la monotonie du store, et sur le système de contraintes, comme l'hypothèse de treillis, qui lui sont propres.

Dans une deuxième partie de la thèse, nous utilisons les propriétés sémantiques des langages concurrents avec contraintes pour prouver dans le chapitre 5 des propriétés de programmes grâce à la sémantique des phases de la logique linéaire (telle qu'elle est décrite par exemple dans [26]). Cette traduction permet de ramener la démonstration de propriétés de sûreté, donc de non-existence de certaines dérivations, à celle, plus simple, de problèmes d'existence, au niveau sémantique; on verra sur des exemples classiques que cette méthode de preuve donne des résultats très satisfaisants. Ensuite nous étudions dans le chapitre 6 comment automatiser, au moins partiellement la méthode exposée précédemment, en tirant partie au mieux de la technologie des contraintes. On aboutit à une première description de cette nouvelle technique de *phase model-checking*. Ce chapitre a donné lieu à une implémentation en GNU-Prolog du système décrit, dont le code est donné en annexe D.

Enfin pour conclure nous exposons brièvement les perspectives qu'ont ouvert ces travaux, les difficultés qui restent à franchir et les solutions qui nous semblent envisageables.

Chapitre 2

Langages CC

"To err is human, but to really foul things up
requires a computer."
– Almanach des fermiers américains, 1978

2.1 Introduction

La programmation concurrente par contraintes, introduite par Saraswat [33] en 1987, est un modèle de calcul concurrent, où les processus (appelés *agents*) communiquent par l'intermédiaire d'un tableau noir (appelé *store*) commun, c'est-à-dire un ensemble de contraintes, donc une information partielle, sur les valeurs des variables.

Nous rappelons ici les définitions classiques des langages CC et fixons les définitions qui nous seront utiles par la suite. Nous insisterons sur les questions d'expressivité de CC et en particulier la séquentialité qui s'exprime avec les opérateurs classiques (*ask*, *tell*, $\|$ et $+$). Nous présenterons aussi le paradigme des CC linéaires, ou LCC, qui étend naturellement celui des CC en se basant sur une logique plus riche: la logique linéaire de Girard [14]. Enfin nous donnons des exemples de programmes et discutons du problème de la vérification de propriétés de ces programmes.

2.2 Définitions et syntaxe

Dans la suite nous noterons X, Y,..., un ensemble de variables et $\vec{x}$ une suite finie de variables. L'ensemble des variables libres apparaissant dans la formule A sera noté $fv(A)$.

Pour un ensemble S, S^* dénote l'ensemble des suites finies d'éléments de S. Pour une relation de transition $\rightarrow$, $\rightarrow^*$ dénote sa fermeture réflexive et transitive.

Définition 2.1 (Système de contraintes) *Un* système de contraintes *est une paire $(\mathcal{C}, \vdash_{\mathcal{C}})$, telle que:*

- *$\mathcal{C}$ est un ensemble de formules (les* contraintes*) construites à partir d'un ensemble V de variables, d'un ensemble Σ de symboles de fonctions et de relations et des opérateurs logiques suivants: 1 ("vrai"), la conjonction $\wedge$ et le quantificateur existentiel $\exists$; $\mathcal{C}$ sera supposé clos par renommage, conjonction et quantification existentielle.*
- *$\Vdash_{\mathcal{C}}$ est un sous-ensemble de $\mathcal{C} \times \mathcal{C}$, qui définit les axiomes non-logiques du système de contraintes. A la place de $(c, d) \in \Vdash_{\mathcal{C}}$, nous noterons $c \Vdash_{\mathcal{C}} d$.*
- *$\vdash_{\mathcal{C}}$ est le plus petit sous-ensemble de $\mathcal{C}^* \times \mathcal{C}$ contenant $\Vdash_{\mathcal{C}}$ et clos par les règles de la Logique Intuitionniste (IL) pour 1, $\wedge$ et $\exists$:*

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

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

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

Dans la suite, $c, d, e \ldots$ dénoterons des contraintes.

Remarque.
Le cadre choisi ici, celui de la logique intuitionniste plutôt que classique, n'est pas essentiel mais simplement suffisant compte tenu de la structure des contraintes qui ne sont construites qu'avec des conjonctions et des implications. Pour les mêmes raisons, et afin de préserver la possibilité d'avoir des contraintes comme `freeze` dans un système de contraintes, nous considérons que la règle d'inférence $\frac{\Gamma \vdash c}{\Gamma \vdash c[t/x]}$ avec $x \notin fv(\Gamma)$ n'est pas indispensable.

Définition 2.2 (Agents) *La syntaxe des* agents *CC est donnée par la grammaire suivante:*

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

Les opérations possibles pour ces agents sont:
- *l'appel de procédure $p(\vec{x})$ (voir la remarque ci-dessous),*

- *l'ajout d'une contrainte c dans le store (agent tell(c) aussi noté plus simplement c), raffinant ainsi l'information partielle sur les variables libres,*
- *la suspension sur une certaine contrainte c: si A est un agent, l'agent* $\forall\vec{x}(ask(c) \to A)$ *(ou plus simplement* $\forall\vec{x}(c \to A)$*) attend, pour exécuter A, que le store contienne suffisamment d'information pour impliquer* $c[\vec{t}/\vec{x}]$*, et lie les variables correspondantes,*
- *la mise en parallèle de deux agents A et B par l'opérateur* $\|$*,*
- *le choix non-déterministe entre deux agents A et B par l'opérateur* $+$ *,*
- *la localisation d'une variable par l'opérateur* $\exists$

Remarque.
Les agents atomiques $p(\vec{x})\ldots$ seront appelés *noms de procédure*, nous supposeront que les arguments dans la suite $\vec{x}$ sont des variables distinctes.

La communication entre agents se déroule par l'intermédiaire du store sous forme d'ajout de contraintes (*tell*) et de suspension sur ces contraintes (*ask*).

L'émission d'information par un *tell* n'est pas bloquante, on a donc à faire à une communication *asynchrone* (voir par exemple [5]).

Les CC étendent la *programmation logique par contraintes* [17, 24] par leur mécanisme de suspension. Ils ont la même sémantique opérationnelle, sauf pour $c \to A$ qui attend que le store implique la contrainte c pour devenir A.

Il n'y a pas, a priori, d'opérateur de séquentialité ";", cependant on verra dans la section 2.4 qu'il peut être encodé à l'aide des opérateurs existants.

L'opérateur *ask* est noté avec un $\forall$ afin de clarifier l'opération de lien des variables de la garde qui est généralement implicite dans CC, on peut donc écrire ainsi dans CC($\mathcal{H}$) le programme (directionnel) de concaténation de deux listes, `L1` et `L2` dans `L3`:

```
concat(L1,L2,L3)=
     (L1=[] → tell(L2=L3)) +
     ∀ E,L (L1=[E|L] → ∃R (tell(L3=[E|R]) || concat(L,L2,R)))
```

On peut cependant noter que, dans un système de contraintes qui n'admet qu'une solution unique à chaque contrainte (comme c'est le cas par exemple sur le domaine de Herbrand), il serait possible de se passer du lien des variables dans la garde en répétant la contrainte gardée dans le corps de la garde. On remplacerait ainsi $\forall\vec{x}(c \to A)$ par $(\exists\vec{x}\ c) \to (\exists\vec{x}\ tell(c) \parallel A)$, mais cette solution, en plus de dissimuler la véritable action du *ask* ne pourrait pas être étendue à LCC (voir section 2.5).

La récursion est obtenue grâce aux déclarations:

Définition 2.3 (Déclarations) *La syntaxe des* déclarations *est donnée par la grammaire suivante:*

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

Définition 2.4 (Programmes) *Un* programme *est une paire $D.A$ où D est une déclaration, close par renommage des variables, et A un agent.*

L'agent A sera appelé *agent initial.*

Les programmes CC sont paramétrés par un système de contraintes, mais en général nous le garderons implicite dans notre présentation, aussi bien dans la relation de transition que dans celle d'implication de contraintes. De même, la déclaration D restera implicite.

Nous ferons l'hypothèse naturelle que dans une déclaration $p(\vec{x}) = A$, toutes les variables de $fv(A)$ apparaissent dans $\vec{x}$. Cela correspond d'ailleurs avec l'interprétation habituelle des clauses de Horn en programmation logique: les variables libres dans le corps d'une clause et n'apparaissant pas dans la tête de celle-ci sont considérées (implicitement dans la syntaxe et explicitement dans la sémantique) comme quantifiées existentiellement.

2.3 Sémantique opérationnelle

2.3.1 Système de transition

La sémantique opérationnelle des programmes CC est définie sur des configurations où le store est distingué des agents:

Définition 2.5 (Configurations) *Une* configuration *est un triplet $(X; c; A)$, où c est une contrainte appelée* store*, A un agent ou $\emptyset$ s'il est vide, et X un ensemble de variables, appelées variables* cachées *de c et A.*

La sémantique opérationnelle est définie par un système de transitions qui ne prend en compte aucune stratégie d'évaluation particulière. Ce système est donné dans le style de la CHAM [3] (voir aussi [29]). Nous distinguerons donc une relation de congruence de la relation de transition.

Définition 2.6 (Congruence) *La* congruence structurelle $\equiv$ *est la plus petite congruence satisfaisant les règles de la table 2.1.*

α-Conversion	$\dfrac{z \notin fv(A)}{\exists y A \equiv \exists z A[z/y]}$
Composition parallèle	$A \parallel B \equiv B \parallel A$ $A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$

TAB. 2.1 – *Congruence structurelle*

Les axiomes d'associativité et de commutativité de la composition parallèle donnent aux agents une structure de multi-ensemble. Désormais nous identifierons donc, par un léger abus de notation, l'agent d'une configuration et le multi-ensemble de ses sous-agents en composition parallèle. Nous noterons $\Gamma, \Delta, ...$ ces multi-ensembles. La congruence s'étend alors de façon canonique aux multi-ensembles: $\Gamma \equiv \Gamma'$ ssi $\Gamma = \{A_1, \ldots, A_n\}$, $\Gamma' = \{A'_1, \ldots, A'_n\}$ et $\forall i = 1, \ldots, n$, $A_i \equiv A'_i$.

Deux configurations seront dites *congrues*, $(X; c; \Gamma) \equiv (X'; c'; \Gamma')$, quand les ensembles X et X' sont égaux et les multi-ensembles d'agents Γ et Γ' congrus.

Définition 2.7 (Transitions) *La relation de* transition $\longrightarrow$ *est la plus petite relation sur les configurations satisfaisant les règles de la table 2.2.*

Dans cette présentation de la relation de transition, il est clair que l'ensemble des variables cachées ne peut que grossir le long d'une dérivation:

Proposition 2.8 *Si* $(X; c; \Gamma) \longrightarrow^* (Y; d; \Delta)$ *alors* $X \subseteq Y$.

Preuve.
Il suffit de raisonner par induction sur la dérivation, puis remarquer que l'ensemble des variables cachées n'est modifié que dans la *localisation* et qu'alors il grossit. □

Les agents et déclarations ne contenant pas + sont appelés *déterministes*. Ce nom se justifie par la proposition suivante:

Proposition 2.9 (Confluence [35]) *Soit une configuration déterministe* κ *avec des déclarations déterministes, si* $\kappa \longrightarrow \kappa_1$ *et* $\kappa \longrightarrow \kappa_2$, *alors il existe une configuration déterministe* κ' *telle que* $\kappa_1 \longrightarrow^* \kappa'$ *et* $\kappa_2 \longrightarrow^* \kappa'$.

Équivalence	$\dfrac{(X;c;\Gamma) \equiv (X';c';\Gamma') \longrightarrow (Y';d';\Delta') \equiv (Y;d;\Delta)}{(X;c;\Gamma) \longrightarrow (Y;d;\Delta)}$
Tell	$(X;c;tell(d),\Gamma) \longrightarrow (X;c \wedge d;\Gamma)$
Ask	$\dfrac{c \vdash_{\mathcal{C}} d[\vec{t}/\vec{x}]}{(X;c;\forall\vec{x}(d \rightarrow A),\Gamma) \longrightarrow (X;c;A[\vec{t}/\vec{x}],\Gamma)}$
Localisation	$\dfrac{y \notin X \cup fv(c,\Gamma)}{(X;c;\exists yA,\Gamma) \longrightarrow (X \cup \{y\};c;A,\Gamma)}$
Appel de procédure	$\dfrac{(p(\vec{y}) = A) \in D}{(X;c;p(\vec{y}),\Gamma) \longrightarrow (X;c;A,\Gamma)}$
Choix aveugle	$(X;c;A+B,\Gamma) \longrightarrow (X;c;A,\Gamma)$ $(X;c;A+B,\Gamma) \longrightarrow (X;c;B,\Gamma)$

TAB. 2.2 – *Relation de transition de CC*

Une autre propriété des programmes CC est que leur exécution est *extensive* (les contraintes ne font que s'accumuler dans le store durant l'exécution) et *monotone*:

Proposition 2.10 (Extensivité [35])
Si $(X;c;\Gamma) \longrightarrow^* (Y;d;\Delta)$ *alors* $\exists Y d \vdash_{\mathcal{C}} \exists X c$.

Proposition 2.11 (Monotonie)
Si $(X;c;\Gamma) \longrightarrow^* (X \cup Y;d;\Delta)$*, alors pour tout ensemble de variables* Z*, tout multi-ensemble d'agents* Σ *et toute contrainte* e*, il existe un renommage* Y' *de* Y *tel que,* $(X \cup Z;c \wedge e;\Gamma,\Sigma) \longrightarrow^* (X \cup Y' \cup Z;d[Y'/Y] \wedge e;\Delta[Y'/Y],\Sigma)$.

Preuve.
On procède par induction sur la longueur de la dérivation. On constate que toutes les règles sauf *équivalence, ask* et *localisation* peuvent être reproduites dans $(X \cup Z;c \wedge e;\Gamma,\Sigma)$ car leurs conditions d'applications ne portent pas sur les variables cachées ni sur le store.

Pour *ask*, il suffit alors de constater que si $c \vdash_{\mathcal{C}} d$ alors $c[Y'/Y] \wedge e \vdash_{\mathcal{C}} d[Y'/Y]$.

Pour la *localisation*, il suffit d'utiliser l'α-conversion de manière à obtenir un y qui ne soit pas dans $fv(e,\Sigma) \cup Z$.

Pour *l'équivalence* il suffit de remarquer que si $d_1 \equiv d_2$ alors $d_1[Y'/Y] \equiv d_2[Y'/Y]$ et la même chose pour Δ_1. □

2.3.2 Observables

La sémantique opérationnelle précise dépend comme d'habitude du choix des observables. Voici celles que nous considéreront:

Définition 2.12 (Observables) *Le* store *d'une configuration* $(X; c; \Gamma)$ *est la contrainte* $\exists X c$. *Nous dirons que* $\exists X d$ *est un* store accessible *de l'agent* A *avec store initial* c, *s'il existe un multi-ensemble d'agents* Γ *tel que* $(\emptyset; c; A) \longrightarrow^* (X; d; \Gamma)$.

Un store succès *(resp. un* succès*) pour un agent* A *et un store initial* c *est une contrainte* $\exists X d$ *(resp. une configuration* $(X; d; \emptyset)$*) telle que* $(\emptyset; c; A) \longrightarrow^* (X; d; \emptyset)$.

Une configuration terminale *pour* A *et le store initial* c *est une configuration* $(X; d; \forall \vec{x}_1(d_1 \rightarrow A_1), \ldots, \forall \vec{x}_n(d_n \rightarrow A_n))$ *telle que* $n \geq 0$, $(\emptyset; c; A) \longrightarrow^* (X; d; \forall \vec{x}_1(d_1 \rightarrow A_1), \ldots, \forall \vec{x}_n(d_n \rightarrow A_n))$ *et* $\forall i, \vec{t}$, $d \not\vdash_{\mathcal{C}} d_i[\vec{t}/\vec{x}_i]$. *Si la configuration n'est pas un succès (i.e.*$n > 0$*), le store* $\exists X d$ *est appelé une* suspension.

Il est facile de vérifier, grâce aux propriétés 2.10 et 2.11, que le comportement opérationnel des programmes CC est complètement caractérisé, sous ces observables, par leur comportement à partir d'un store initial vide:

Proposition 2.13 *Soit* A *un agent et* c *une contrainte.*

Les stores accessibles de A *avec un store initial* c *sont les conjonctions de* c *et des stores accessibles de* $A \parallel tell(c)$ *avec un store initial vide.*

Ses succès et ses configurations terminales sont ceux de $A \parallel tell(c)$ *dans le store initial vide.*

Preuve.
On raisonne par induction sur la derivation en utilisant la monotonie (prop. 2.11) jusqu'à ce que l'agent $tell(c)$ ajoute vraiment sa contrainte au store. □

La sémantique opérationnelle peut donc être définie avec le store initial vide (i.e. la contrainte "vrai" notée 1):

Définition 2.14 (Sémantique opérationnelle)
$\mathcal{O}^{store}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid \exists \Gamma,\ (\emptyset; 1; A) \longrightarrow^* (X; d; \Gamma)\}$
$\mathcal{O}^{term.}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid \exists \Gamma,\ (\emptyset; 1; A) \longrightarrow^* (X; d; \Gamma) \not\longrightarrow\}$
$\mathcal{O}^{succès}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid (\emptyset; 1; A) \longrightarrow^* (X; d; \emptyset)\}$

où $\kappa \not\longrightarrow$ *dénote l'absence de dérivation partant de* κ.

Remarque.
Dans le système de transition, nous avons choisi la règle de choix aveugle: l'agent non-déterministe $A + B$ peut se comporter comme A ou comme B. Si l'on remplace ce choix "interne" par le choix gardé:

$$\frac{(X;c;A,\Gamma) \longrightarrow (Y;d;\Delta)}{(X;c;A+B,\Gamma) \longrightarrow (Y;d;\Delta)} \quad et \quad \frac{(X;c;B,\Gamma) \longrightarrow (Y;d;\Delta)}{(X;c;A+B,\Gamma) \longrightarrow (Y;d;\Delta)}$$

les suspensions d'un programme changent bien évidement. Par exemple si $\kappa = (\emptyset;c;(c \rightarrow tell(1)) + (d \rightarrow tell(1)))$ avec $c \not\vdash_{\mathcal{C}} d$, alors pour le choix aveugle, κ a deux dérivations possibles: $\kappa \longrightarrow (\emptyset;c;c \rightarrow tell(1)) \longrightarrow (\emptyset;c;tell(1)) \longrightarrow (\emptyset;c;\emptyset)$ et $\kappa \longrightarrow (\emptyset;c;d \rightarrow tell(1)) \not\longrightarrow$, alors que la deuxième dérivation n'est pas possible avec le choix gardé. Il est cependant intéressant de noter que les succès et stores accessibles restent inchangés dans les deux interprétations. Les résultats de cette thèse qui ne concernent que ces deux types d'observables restent donc valides pour le choix gardé.

Proposition 2.15 *Soient $\mathcal{O}^{store}_{gardé}$ et $\mathcal{O}^{succès}_{gardé}$ la sémantique opérationnelle définie comme ci-dessus mais avec les règles du choix gardé. Pour tout agent CC, A, on a:*
$\mathcal{O}^{store}(\mathcal{C},\mathcal{D}.A) = \mathcal{O}^{store}_{gardé}(\mathcal{C},\mathcal{D}.A)$ *et* $\mathcal{O}^{succès}(\mathcal{C},\mathcal{D}.A) = \mathcal{O}^{succès}_{gardé}(\mathcal{C},\mathcal{D}.A)$

Preuve.
La preuve se fait par une simple induction: si l'on note $\longrightarrow_{gardé}$ la relation de transition avec les règles du choix gardé, considérons une dérivation pour $\longrightarrow$, elle ne diverges d'une dérivation pour $\longrightarrow_{gardé}$ que quand elle s'arrête après une transition utilisant une règle de choix aveugle, mais alors: (1) le store n'a pas changé, donc les stores accessibles ne changent pas et (2) la nouvelle configuration terminale n'est pas un succès, et la configuration à l'étape du choix, qui aurait pu être terminale, n'en était pas un non plus. □

2.4 Séquentialité

Bien que les langages CC ne possèdent pas d'opérateur de séquentialité, il s'avère que cet opérateur peut être *codé* dans CC sans aucune modification:

Proposition 2.16 *Soit D.A un programme de $CC_{seq}(\mathcal{C})$, c'est à dire CC étendu de l'opérateur $\bullet$ défini par les règles suivantes:*

$$\frac{(X;c;A) \longrightarrow (Y;d;B)}{(X;c;A \bullet C,\Gamma) \longrightarrow (Y;d;B \bullet C,\Gamma)} \qquad (X;c;\emptyset \bullet A) \longrightarrow (X;c;A)$$

Il existe un système de contraintes $\mathcal{C}^\bullet$ et un programme $D^\bullet.A^\bullet$ de $CC(\mathcal{C}^\bullet)$ dont les observables permettent de caractériser entièrement celles de $D.A$ dans $CC_{seq}(\mathcal{C})$.

Preuve.
Soit *ok* un *nouveau* symbole de relation d'arité un. $\mathcal{C}^\bullet$ est le système de contraintes $\mathcal{C}$ auquel on a ajouté *ok* sans aucun axiome non-logique. Le programme $D^\bullet.A^\bullet$ est défini par la traduction suivante:

$$\begin{aligned}
tell(c)^\bullet_x &= tell(c \wedge ok(x)) \\
p(\vec{y})^\bullet_x &= p^\bullet(x, \vec{y}) \\
(A \parallel B)^\bullet_x &= \exists y, z(A^\bullet_y \parallel B^\bullet_z \parallel (ok(y) \wedge ok(z)) \rightarrow ok(x)) \\
(A + B)^\bullet_x &= A^\bullet_x + B^\bullet_x \\
(\forall \vec{y}(c \rightarrow A))^\bullet_x &= \forall \vec{z}(c[\vec{z}/\vec{y}] \rightarrow A[\vec{z}/\vec{y}]^\bullet_x) \; avec \; x \notin \vec{z} \\
(\exists y A)^\bullet_x &= \exists z A[z/y]^\bullet_x \; avec \; z \neq x \\
(A \bullet B)^\bullet_x &= \exists y(A^\bullet_y \parallel ok(y) \rightarrow B^\bullet_x) \\
(p(\vec{y}) = A)^\bullet &= p^\bullet(x, \vec{y}) = A^\bullet_x \\
A^\bullet &= \exists x A^\bullet_x
\end{aligned}$$

Il est immédiat de vérifier (par une induction sur la dérivation) qu'une contrainte $ok(x)$ est ajoutée au store si et seulement si l'agent correspondant $A^\bullet_x$ a atteint un succès.

Or les arbres de dérivation de $D.A$ et de $D^\bullet.A^\bullet$ ne diffèrent que par des chaînes ajoutées à certaines feuilles. En effet les seuls agents ajoutés par la traduction sont de la forme $ok(y) \wedge ok(z) \rightarrow ok(x)$ et par la proposition de monotonie 2.11 (qui reste valable avec l'ajout de $\bullet$) les éventuelles règles **Ask** peuvent être considérées comme ayant lieu à la fin.

De plus, les stores au cours de la dérivation de $D^\bullet.A^\bullet$ ne diffèrent que très peu de leur store correspondant dans la dérivation de $D.A$; en effet ils ne comportent en plus que des contraintes de la forme $ok(x)$ introduites par un *tell*.

Comme les prolongements de dérivation ne font qu'ajouter d'autres contraintes $ok(x)$ au store courant, on a donc:

$$Ost(\mathcal{C}, D.A) = \pi_{\mathcal{C}} Ost(\mathcal{C}', D^\bullet.A^\bullet)$$

où $\pi_{\mathcal{C}}$ est l'opération qui consiste à projeter sur $\mathcal{C}$ une contrainte de $\mathcal{C}'$ en lui enlevant tous ses composants de la forme $ok(x)$.

Or si on obtient un succès pour une dérivation de $D.A$ nous avons vu que la contrainte *ok* correspondante était ajoutée au store, et réciproquement, par conséquent on a:

$$Osuc(\mathcal{C}, D.A) = \pi_{\mathcal{C}} Osuc(\mathcal{C}', D^\bullet.A^\bullet)$$

De même, si $D.A$ suspend, alors $D^\bullet.A^\bullet$ suspend sur le même *ask*, et comme il n'y a jamais de suspension sur le *ask* ajouté en fin de branche sans qu'il y en ait une avant (expliquant ainsi la non-production d'un *ok*) on a donc aussi les mêmes suspensions et donc:

$$Ostt(\mathcal{C}, D.A) = \pi_{\mathcal{C}} Ostt(\mathcal{C}', D^\bullet.A^\bullet)$$

□

2.5 CC Linéaires

Il y a en fait plusieurs raisons qui ont mené à l'utilisation de la logique linéaire comme base du système de contraintes, et donc à la naissance des CC linéaires:

- tout d'abord, malgré l'expressivité importante de CC, qui par exemple permet de coder la séquentialité comme nous l'avons vu dans la section précédente, on peut vouloir encore augmenter le pouvoir d'expression de ces langages, en particulier pour permettre un meilleur contrôle de la concurrence (voir par exemple [36]). Une option est de permettre la consommation de certaines contraintes, ce que Saraswat et Lincoln ont compris dès 1992 [34] et qui fut ensuite poussé plus loin dans [4, 37]. Les contraintes y sont alors naturellement des formules de la logique linéaire.
- Un autre motif qui pousse à se diriger vers LCC nous est donné par la sémantique, en effet nous verrons dans le chapitre 3 que la logique linéaire s'impose comme un outil très puissant pour étudier CC, en permettant de caractériser des observables plus fines que la logique intuitionniste.

Nous présentons donc ici les CC linéaires comme cette suite naturelle des CC; nous en donnons des exemples et chercherons dans les chapitres qui suivent à prouver des propriétés de tels programmes.

2.5.1 Syntaxe

Comme dans les cas des CC nous allons définir le système de contraintes, les agents et configurations, et le système de transition. La différence essentielle sera la nature linéaire des contraintes et donc la consommation d'information par l'opérateur *ask*.

Définition 2.17 (Système de contraintes Linéaire) *Un* système de contraintes linéaire *est une paire* $(\mathcal{C}, \vdash_{\mathcal{C}})$*, telle que:*

- $\mathcal{C}$ *est un ensemble de formules (les* contraintes linéaires*) construites à partir d'un ensemble* V *de variables, d'un ensemble* Σ *de symboles de fonction et de relation et des opérateurs logiques suivants:* 1 *(i.e. "vrai"), la conjonction* $\otimes$*, l'exponentiel* $!$ *et le quantificateur existentiel* $\exists$*;* $\mathcal{C}$ *sera supposé clos par renommage,* $\otimes$*,* $!$ *et* $\exists$*.*
- $\Vdash_{\mathcal{C}}$ *est un sous-ensemble de* $\mathcal{C} \times \mathcal{C}$*, qui définit les axiomes non-logiques du système de contraintes.*
- $\vdash_{\mathcal{C}}$ *est le plus petit sous-ensemble de* $\mathcal{C}^* \times \mathcal{C}$ *contenant* $\Vdash_{\mathcal{C}}$ *et clos par les règles de la Logique Linéaire Intuitionniste (ILL) pour* 1*,* $\otimes$*,* $!$ *et* $\exists$ *(voir aussi l'annexe A):*

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

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

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

Remarque.
Les systèmes de contraintes (classiques) de la section précédente sont un cas particulier que l'on peut retrouver en utilisant la traduction classique de IL dans ILL qui consiste à tout mettre sous un ! [14]. L'utilisation du ! est limitée aux contraintes et non pas aux agents, il est donc à différentier de l'opérateur de réplication des calculs de processus (comme par exemple le π-calcul [25]) qui permettrait la duplication d'un agent (remplacée dans LCC par l'appel de procédure), mais pas l'effacement.

On peut définir une classe de contraintes particulières, appelées *contraintes de synchronisation* [32] et représentant les contraintes atomiques qui n'apparaissent dans aucun axiome non-logique. Nous verrons plus loin et dans le chapitre 3 que ces contraintes bénéficient de nombreuses bonnes propriétés sémantiques que nous étendrons aux autres contraintes dans le chapitre 4.

La définition des *agents*, *déclarations* et *configurations* est la même que pour CC (section 2.2).

Définition 2.18 (Système de transition) *La* congruence structurelle $\equiv$ *reste identique à celle de CC (section 2.3) et la* relation de transition *est définie par les mêmes règles, excepté pour* **LinearTell** *et* **LinearAsk** *qui remplacent* **Tell** *et* **Ask***:*

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

TAB. 2.3 – *Relation de transition de LCC*

Le calcul est intrinsèquement non-déterministe, même en l'absence de + puisque plusieurs contraintes peuvent satisfaire la garde d'un ask et donc mener à des résidus différents. Pour les *contraintes de synchronisation* par contre, *ask* (sans $\forall$) et *tell* deviennent des opérateurs de manipulation élémentaire de multi-ensemble et sont donc déterministes. Des exemples d'utilisation de ces contraintes sont donnés dans la section 2.5.3.

La nature linéaire des contraintes nous oblige à reformuler la définition des configurations terminales.

Définition 2.19 (Observables) *Les* succès *et* stores accessibles *sont définis comme pour CC.*

Une configuration terminale *pour A et le store initial c est une configuration $(X; d; \forall \vec{x}_1(d_1 \to A_1), \ldots, \forall \vec{x}_n(d_n \to A_n)$ telle que $n \geq 0$, $(\emptyset; c; A) \longrightarrow^* (X; d; \forall \vec{x}_1(d_1 \to A_1), \ldots, \forall \vec{x}_n(d_n \to A_n))$ et $\forall i, \vec{t}, d \not\vdash_{\mathcal{C}} d_i[\vec{t}/\vec{x}_i] \otimes \top$.*

On a seulement introduit la constante $\top$ pour refléter le changement de condition dans la règle **LinearAsk**: la garde se débloque s'il existe e tel que $c \vdash_{\mathcal{C}} d \otimes e$ on aurait alors $c \vdash_{\mathcal{C}} d \otimes \top$ d'après la règle pour $\top$.

Ce qui rend LCC beaucoup plus expressif que CC est la perte d'*extensivité*, donc la possibilité d'exprimer des évolutions non-monotones du store. Il est cependant important de noter que la *monotonie des transitions* est conservée.

Proposition 2.20 (Monotonie) *Si $(X; c; \Gamma) \longrightarrow^*_{LCC} (X \cup Y; d; \Delta)$, alors pour tout ensemble de variables Z, tout multi-ensemble d'agents Σ et toute contrainte e, il existe un renommage Y' de Y tel que $(X \cup Z; c \otimes e; \Gamma, \Sigma) \longrightarrow^*_{LCC} (X \cup Y' \cup Z; d[Y'/Y] \otimes e; \Delta[Y'/Y], \Sigma)$.*

Preuve.
La preuve suit la même induction que pour la proposition 2.11 concernant la monotonie de CC, puisque les deux seules règles qui ont changé ne modifient pas l'ensemble des variables cachées et n'en dépendent pas pour leur exécution. □

Comme dans la section précédente les observables à partir d'un store initial vide suffisent donc à retrouver celles provenant d'un calcul avec un store initial arbitraire. L'argument ne change que pour l'observation des stores accessibles:

Proposition 2.21 *Soit $\mathcal{C}$ un système de contraintes, et $\mathcal{C}'$ le système de contraintes obtenu en ajoutant à $\mathcal{C}$ un* nouveau *symbole de contraintes d. L'ensemble des stores accessibles de $(\emptyset; c; A)$ dans $\mathcal{C}$ est $\{e \in \mathcal{C} \mid e \otimes d \in \mathcal{O}^{store}_{LCC}(\mathcal{C}', \mathcal{D}.(tell(c \otimes d) \parallel (d \to A)))\}$.*

Preuve.
Comme d est *nouveau*, les seules transitions possibles sont:
$(\emptyset; 1; tell(c \otimes d \otimes d) \parallel (d \to A)) \longrightarrow_{LCC} (\emptyset; c \otimes d \otimes d; d \to A) \longrightarrow_{LCC} (\emptyset; c \otimes d; A) \longrightarrow_{LCC} \cdots$.

Les stores accessibles de $(\emptyset; c; A)$ sont donc les $e \in \mathcal{C}$ (i.e. ne contenant pas d) tels que $e \otimes d$ est accessible de $(\emptyset; 1; tell(c \otimes d \otimes d) \parallel (d \to A))$[1]. □

Définition 2.22 (Sémantique opérationnelle)
$\mathcal{O}^{store}_{LCC}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid \exists \Gamma,\ (\emptyset; 1; A) \longrightarrow^*_{LCC} (X; d; \Gamma)\}$
$\mathcal{O}^{term.}_{LCC}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid \exists \Gamma,\ (\emptyset; 1; A) \longrightarrow^*_{LCC} (X; d; \Gamma) \not\longrightarrow_{LCC}\}$
$\mathcal{O}^{succès}_{LCC}(\mathcal{C}, \mathcal{D}.A) = \{\exists X d \in \mathcal{C} \mid (\emptyset; 1; A) \longrightarrow^*_{LCC} (X; d; \emptyset)\}$

2.5.2 Traduction de CC dans LCC

Le paradigme LCC est une extension des CC, dans lequel on peut traduire simplement et fidèlement les CC monotones en respectant l'observation des stores et des succès [12]:

Définition 2.23 *Soit $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes. Sa* traduction *est le système de contraintes linéaire $(\mathcal{C}^\circ, \Vdash^\circ_{\mathcal{C}})$, défini comme suit avec la traduction*

1. On peut noter que remplacer dans le codage $c \otimes d$ par $c \otimes d \otimes d$ ne permet pas de décider si 1 est accessible ou pas.

des agents:

$$c^\circ = !c, \text{ si } c \text{ est une contrainte atomique}$$

$$(c \wedge d)^\circ = c^\circ \otimes d^\circ \qquad (\exists x c)^\circ = \exists x c^\circ$$
$$tell(c)^\circ = tell(c^\circ) \qquad p(\vec{x})^\circ = p(\vec{x})$$
$$(A \parallel B)^\circ = A^\circ \parallel B^\circ \qquad (A + B)^\circ = A^\circ + B^\circ$$
$$(\forall \vec{x}(c \to A))^\circ = \forall \vec{x}(c^\circ \to A^\circ) \qquad (\exists x A)^\circ = \exists x A^\circ$$

$\mathcal{C}^\circ = \{c^\circ \mid c \in \mathcal{C}\}$.
$\Vdash_{\mathcal{C}^\circ}$ *est définie par* $c \Vdash_{\mathcal{C}} d$ *ssi* $c^\circ \Vdash_{\mathcal{C}^\circ} d^\circ$
Cette traduction s'étend naturellement aux configurations.

Pour les contraintes, cette traduction est une traduction bien connue de IL dans ILL [14, p.81], d'où:

Proposition 2.24 *Soient* c *et* d *des contraintes de* $\mathcal{C}$*:* $c \vdash_{\mathcal{C}} d$ *ssi* $c^\circ \vdash_{\mathcal{C}^\circ} d^\circ$.

On peut alors vérifier que le comportement des configurations traduites est bien celui qui est attendu:

Proposition 2.25 *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *des configurations CC:*
(i) $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *ssi* $(X; c^\circ; \Gamma^\circ) \equiv^\circ (Y; d^\circ; \Delta^\circ)$*;*
(ii) si $(X; c; \Gamma) \longrightarrow (Y; d; \Delta)$ *alors* $(X; c^\circ; \Gamma^\circ) \longrightarrow_{LCC} (Y; d^\circ; \Delta^\circ)$*;*
(iii) si $(X; c^\circ; \Gamma^\circ) \longrightarrow_{LCC} (Y; d^\circ; \Delta^\circ)$ *alors* $(X; c; \Gamma) \longrightarrow (Y; e; \Delta)$*, avec* $e \vdash_{\mathcal{C}} d$.

Preuve.
(i) est évident.

Pour (ii), procédons par induction sur $\longrightarrow$, seul le cas de la règle **Ask** est intéressant: supposons

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

utilisant la relation $c \vdash_{\mathcal{C}} d[\vec{t}/\vec{x}]$. On a donc $c \vdash_{\mathcal{C}} c \wedge d[\vec{t}/\vec{x}]$, et d'après la proposition 2.24, $c^\circ \vdash_{\mathcal{C}^\circ} (c \wedge d[\vec{t}/\vec{x}])^\circ = c^\circ \otimes d[\vec{t}/\vec{x}]^\circ$. D'où

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

cqfd.

Pour (iii), procédons par induction sur $\longrightarrow_{LCC}$, et à nouveau, le seul cas intéressant est celui de la règle **LinearAsk**: supposons

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

utilisant la relation $c^\circ \vdash_{\mathcal{C}^\circ} d[\vec{t}/\vec{x}]^\circ \otimes e^\circ = (d[\vec{t}/\vec{x}] \wedge e)^\circ$. D'après 2.24, $c \vdash_{\mathcal{C}} d[\vec{t}/\vec{x}] \wedge e \vdash_{\mathcal{C}} d[\vec{t}/\vec{x}]$, donc

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

et $c \vdash_{\mathcal{C}} e$. □

(i) et (ii) prouvent que la traduction ci-dessus est correcte vis à vis de l'observation des stores et des succès d'un calcul CC.

2.5.3 Exemple de programme LCC

Un test classique de l'expressivité des langages concurrents est le problème du dîner des philosophes: `N` philosophes sont assis autour d'une table ronde, entre chaque paire est posée une fourchette. Les philosophes alternent des périodes de pensée profonde avec des périodes de faim, et dans ce dernier cas ils ont besoin des deux fourchettes qui les entourent pour manger (des baguettes seraient certes plus vraisemblables).

Comme suggéré dans [4], ce problème a une solution simple et élégante en LCC.

Les contraintes atomiques sont: `fork(I), eat(I,N)` pour `I,N` $\in \mathbb{N}$, et `N=M, N≠M` pour `N,M` $\in \mathbb{N}$. Le système de contraintes linéaires est une combinaison de la traduction de la théorie de l'égalité pour $(\mathbb{N}, +)$, et des contraintes linéaires `fork` et `eat` avec pour seuls axiomes non logiques le schéma correspondant à l'axiome d'égalité: $c(\vec{x}) \otimes !(\vec{x} = \vec{y}) \Vdash c(\vec{y})$ pour toute contrainte c.

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

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

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

Une exécution avec l'agent initial `init(5)`, par exemple, va mettre en place les philosophes et les fourchettes, puis la séquence (infinie) des stores le long d'une dérivation montrera les périodes où les philosophes mangent, compte tenu du scheduling (ordonnancement d'exécution) des agents en composition parallèle.

Contrairement à un programme CC classique, les structures de données impératives sont ici directement codées par des contraintes linéaires au lieu de streams [33]. On peut aussi noter que par rapport à des langages comme Linda, il est ici inutile d'avoir des "tickets"[7] puisque l'opérateur *ask* permet de consommer atomiquement le tenseur de deux fourchettes.

Ce programme vérifie certaines propriétés de vivacité et de sûreté, nous verrons une méthode de preuve de ces propriétés utilisant la sémantique des phases de la Logique Linéaire dans le chapitre 5.

Un autre exemple illustrant le gain d'expressivité qu'apporte LCC est donné dans l'annexe B (il est un peu long pour figurer ici). Il s'agit de la reconstruction d'un solveur sur les domaines finis à la manière CLP(FD) et sans les *indexicaux* de [15]. Ce programme, ainsi que tous les exemples donnés dans cette thèse ont été testés grâce à une implémentation prototype de LCC que nous avons réalisé au dessus de GNU-Prolog.

D'autres exemples enfin peuvent être trouvés dans la thèse de Vincent Schächter [36], comme par exemple un solveur basé sur le simplexe.

Chapitre 3

Sémantiques

"It might look like I'm doing nothing,
but at the cellular level I'm really quite busy."
– Dilbert (Scott Adams)

3.1 Introduction

Les langages CC étant issus de la programmation logique, il est naturel de chercher à connecter leurs aspects opérationnels avec une sémantique formelle permettant de raisonner sur les programmes à différents niveaux d'abstraction. L'évolution monotone du store pendant un calcul CC a permis de donner une sémantique dénotationnelle simple et élégante aux programmes CC, sous la forme d'opérateurs de fermeture sur les contraintes [35, 18]. Cette sémantique a été utilisée, par exemple dans [10] pour obtenir un système de preuve de certaines propriétés de programmes. Nous verrons dans la section 3.5 quelles sont les résultats les plus précis obtenus avec cette approche.

Cependant, même si l'ajout du *ask* a brisé la sémantique logique simple de la programmation logique, nous rappelons ici comment en logique intuitionniste puis dans la logique linéaire de Girard [14], il est possible de faire à nouveau le lien CC-logique. Une partie des résultats présentés ici apparaissaient déjà dans [22, 34], mais n'ont été formalisés complètement que dans [30] et [12]. cette approche couvre une plus grande variété d'observables et s'adapte naturellement au cadre LCC, mais comporte des limites que nous exposerons au fur et à mesure.

3.2 Caractérisation des stores CC en logique intuitionniste

Soient $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes et $\mathcal{D}$ des déclarations,

Définition 3.1 *On peut traduire les agents CC déterministes en formules de la logique intuitionniste (IL) de la manière suivante:*

$$tell(c)^\dagger = c$$
$$p(\vec{x})^\dagger = p(\vec{x}) \qquad (\exists x A)^\dagger = \exists x A^\dagger$$
$$(\forall \vec{x}(c \to A))^\dagger = \forall \vec{x}(c \Rightarrow A^\dagger) \quad (A \parallel B)^\dagger = A^\dagger \wedge B^\dagger$$

Cette traduction s'étend aux multi-ensembles d'agents: Si Γ *est le multi-ensemble* $\{A_1 \dots A_n\}$*, alors définissons* $\Gamma^\dagger = A_1^\dagger \wedge \dots \wedge A_n^\dagger$*. Et* $\Gamma^\dagger = 1$ *si* $\Gamma = \emptyset$*.*

La traduction d'une configuration $(X; c; \Gamma)^\dagger$ *est la formule* $\exists X(c \wedge \Gamma^\dagger)$*.*

notons $\mathrm{IL}(\mathcal{C}, \mathcal{D})$ le système de déduction obtenu en ajoutant à IL les axiomes non-logiques suivants:

- $c \vdash d$ pour tout $c \Vdash_{\mathcal{C}} d$ de $\Vdash_{\mathcal{C}}$,
- $p(\vec{x}) \vdash A^\dagger$ pour toute déclaration $p(\vec{x}) = A$ de $\mathcal{D}$.

$\dashv\vdash$ dénote l'équivalence logique.

Théorème 3.2 (Correction [12]) *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *deux configurations CC déterministes.*
Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^\dagger \dashv\vdash_{IL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\dagger$*.*
Si $(X; c; \Gamma) \longrightarrow^* (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\dagger$*.*

La réciproque est vraie pour l'observation des *stores*:

Théorème 3.3 (Observation des stores pour CC déterministe [12])

Soit S *un ensemble de contraintes,et* A *un agent CC déterministe, notons* $\downarrow S = \{c \in \mathcal{C} \mid \exists d \in S, d \vdash_{\mathcal{C}} c\}$ *et* $\mathcal{L}^{store}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} c\}$*, on a:*

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

Le cas non-déterministe n'est pas évident dans IL, en effet traduire $+$ par la disjonction $\vee$ ne convient pas pour la correction $(A \vee B \not\vdash A)$ et traduire par la conjonction $\wedge$, ne permet pas de garder la complétude sur l'observation des stores ($c \wedge d$ n'est pas dans $\downarrow \mathcal{O}^{store}(tell(c) + tell(d))$).

3.3 Caractérisation des stores et succès en logique linéaire intuitionniste

Bien que l'observation des stores soit importante, elle ne représente qu'un des aspects du comportement opérationnel des programmes CC. Par exemple, si l'on considère les trois programmes suivants:

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

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

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

Ils ont les mêmes stores ($x \geq 1$) et sont donc équivalents vis à vis de cette observable, cependant l'un termine avec un succès, l'autre boucle et le troisième suspend.

Or, comme le montrent les contre-exemples suivants, IL ne permet de caractériser ni les succès, ni les suspensions:

- $\dashv$: Il est en général faux que $A \dashv B$ (où B est un succès ou une suspension) implique $(\emptyset; 1; A) \longrightarrow_{LCC} (\emptyset; 1; B)$. Par exemple $c \rightarrow d \dashv d$ mais $c \rightarrow d$ suspend dans le store initial vide. Pour le cas où B est une suspension, par exemple $d \parallel (c \rightarrow d)$ avec d n'impliquant pas c, on a $d \dashv d \wedge (c \rightarrow d)$ mais $tell(d)$ ne se réduit pas vers cette suspension.
- $\vdash$: On rencontre des problèmes similaires avec $\vdash$. On a $d \wedge (c \Rightarrow A) \vdash d$ mais $d \parallel (c \rightarrow A)$ suspend si $d \not\vdash c$. De plus $d \wedge (d \Rightarrow e) \vdash d \Rightarrow e$, mais $d \parallel (d \rightarrow e)$ a un succès $(d \wedge e)$ et pas de suspensions.
- $\dashv\vdash$: L'équivalence $\dashv\vdash$ pose les mêmes problèmes, par exemple on remarque que $d \wedge (c \Rightarrow d) \dashv\vdash d$, mais on ne peut rien en déduire sur le comportement opérationnel des agents $tell(d)$ et $d \parallel (c \rightarrow d)$ en supposant que d n'implique pas c.

L'obstacle principal est en fait la règle (structurelle) d'affaiblissement:

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

La Logique Linéaire (LL) de Girard [14] raffinant la logique classique, en particulier au niveau des règles structurelles d'affaiblissement et de contraction, il est naturel d'interpréter les programmes CC dans cette logique plus fine. C'est d'ailleurs cette démarche qui a conduit à la création des langages LCC qu'on peut bien sur interpréter par la même occasion.

Fixons un système de contraintes linéaire $(\mathcal{C}, \Vdash_{\mathcal{C}})$ et des déclarations $\mathcal{D}$.

Définition 3.4 *On traduit les agents LCC en formules de la logique linéaire (intuitionniste) de la manière suivante:*

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

Si Γ *est le multi-ensemble* $(A_1 \ldots A_n)$*, on définit* $\Gamma^{\ddagger} = A_1^{\ddagger} \otimes \cdots \otimes A_n^{\ddagger}$*. Si* $\Gamma = \emptyset$ *alors* $\Gamma^{\ddagger} = 1$.

La traduction $(X; c; \Gamma)^{\ddagger}$ *d'une configuration, est la formule* $\exists X(c \otimes \Gamma^{\ddagger})$.

ILL$(\mathcal{C}, \mathcal{D})$ dénote le système de déduction obtenu en ajoutant à ILL les axiomes non-logiques:

- $c \vdash d$ pour tout $c \Vdash_{\mathcal{C}} d$ de $\Vdash_{\mathcal{C}}$,
- $p(\vec{x}) \vdash A^{\ddagger}$ pour toute déclaration $p(\vec{x}) = A$ de $\mathcal{D}$.

$\dashv\vdash$ dénotera l'équivalence logique.

Théorème 3.5 (Correction [12]) *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *deux configurations LCC.*

Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^{\ddagger} \dashv\vdash_{ILL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^{\ddagger}$.

Si $(X; c; \Gamma) \longrightarrow^{*}_{LCC} (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^{\ddagger}$.

Réciproquement, on peut caractériser l'observation des succès, même en présence de l'opérateur de choix +:

Théorème 3.6 (Observation des succès [12]) *Soit A un agent LCC et c une contrainte linéaire. Définissons* $\mathcal{LL}^{succès}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} | A^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c\}$*. On a*

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

Soit S un ensemble de contraintes linéaires, notons $\Downarrow S = \{c \in \mathcal{C} \mid \exists d \in S, d \vdash c \otimes \top\}$.

Théorème 3.7 (Observation des stores [12]) *Soit A un agent LCC et c une contrainte linéaire. Définissons* $\mathcal{LL}^{store}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} | A^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} c \otimes \top\}$*. On a*

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

Grâce à la traduction de CC dans LCC (section 2.5.2), cette caractérisation des stores et des succès en logique linéaire est aussi valable pour CC.

3.4 Caractérisation des suspensions en logique non-commutative

Même en logique linéaire, on n'arrive toujours pas à caractériser les suspensions, l'un des obstacles étant l'adjonction entre $\otimes$ et $\multimap$ qui peut générer de *fausses suspensions* comme par exemple dans le cas $c \otimes (c \multimap d) \vdash c \multimap (c \otimes d)$. Pour supprimer cette adjonction, on peut utiliser la Logique Non-commutative (NL) de Ruet et Abrusci [31] introduite à cet effet dans [30] (voir aussi l'annexe C).

La traduction des agents LCC dans le fragment intuitionniste de NL ne diffère de la précédente que par la traduction du *ask*:

$(c \to A)^\Diamond = c \bullet\!\!\to (\diamond \otimes A)$

où $\diamond$ est une nouvelle formule atomique, introduite pour éviter la *composition des suspensions* $(d \otimes (c \bullet\!\!\to d) \otimes (d \bullet\!\!\to A) \vdash d \otimes (c \bullet\!\!\to A))$. On ne considère dans ce cadre que les agents LCC sans $\forall$, mais nous avons vu dans la section 2.2 que cela contient déjà tout CC.

Théorème 3.8 (Correction [30]) *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *deux configurations LCC.*

Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^\Diamond \dashv\vdash_{INL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\Diamond$.

Si $(X; c; \Gamma) \longrightarrow^*_{LCC} (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^\Diamond \vdash_{INL(\mathcal{C},\mathcal{D})} (Y; d; \Delta)^\Diamond$.

Réciproquement, on peut toujours caractériser l'observation des succès:

Théorème 3.9 (Observation des succès [30]) *Soit* A *un agent LCC et* c *une contrainte linéaire. Définissons* $\mathcal{NL}^{succès}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} | A^\Diamond \vdash_{INL(\mathcal{C},\mathcal{D})} \vec{\diamond} \otimes c\}$. *On a*

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

Théorème 3.10 (Observation des suspensions [30]) *Soient* $A, A_1, \ldots, A_n$ *des agents LCC, c une contrainte linéaire,* $d_1, \ldots, d_n$ *des contraintes de synchronisation telles que* $\forall i, c \not\vdash d_i \otimes \top$. *Si* $A^\Diamond \vdash_{INL(\mathcal{C},\mathcal{D})} \vec{\diamond} \otimes c \otimes (d_1 \bullet\!\!\to A_1) \otimes \ldots \otimes (d_n \bullet\!\!\to A_n)\}$, *alors* $(\emptyset; 1; A)$ *suspend avec le store c sur les contraintes* $d_1, \ldots, d_n$.

Remarque.
Il est important de noter ici que ce résultat s'appuyant sur les résultats de la section précédente, c'est à dire sur des caractérisations modulo clôture vers le bas, il ne s'applique qu'au cas de contraintes de synchronisation, c'est à dire de contraintes sans aucun axiome non-logique.

Une caractérisation plus précise, évitant la clôture, permettrait donc, en plus de ses apports propres pour mieux caractériser stores et succès, de généraliser le résultat de caractérisation des suspensions à des contraintes quelconques. C'est ce que nous montrerons dans le chapitre 4.

3.5 Sémantique dénotationnelle

Jusqu'à présent nous avons rappelé les différents résultats obtenus par l'approche *sémantique logique* de CC qui permet donc de caractériser les stores, succès et suspensions des agents CC (et même LCC) sous réserve de restrictions sur la structure des agents dans le cas des suspensions, et à une clôture près ($\downarrow$ ou $\Downarrow$) dans les autres cas.

Intéressons-nous désormais aux résultats obtenus par l'autre approche principale dans ce domaine, *la sémantique dénotationnelle*. Un agent CC déterministe y est vu comme une fonction des contraintes dans les contraintes, associant au store initial le store après exécution de l'agent. Cela aboutit à une sémantique dénotationnelle très simple définie par des opérateurs de clôture. En effet un agent CC ne peut qu'ajouter une contrainte au store (extensivité), exécuter une deuxième fois un même agent ne changera pas le store (idempotence), enfin plus le store initial contient d'information, plus le store final en contiendra (monotonie). En représentant un opérateur de clôture par l'ensemble de ses points fixes on obtient [35]:

$$
\begin{array}{ll}
[\![D.tell(c)]\!](e) & = \uparrow c \\
[\![D.c \rightarrow A]\!](e) & = (\uparrow c \cap [\![D.A]\!](e)) \cup \overline{\uparrow c} \\
[\![D.A \parallel B]\!](e) & = [\![D.A]\!](e) \cap [\![D.B]\!](e) \\
[\![\exists x A]\!](e) & = \exists_x [\![D.A]\!](e) \\
[\![D.p(\vec{x})]\!](e) & = \mu\Psi \text{ où } \Psi(f) = [\![D \setminus \{p\}.A]\!](e\{f/p\}) \text{ si } p = A \in D \\
[\![D.p(\vec{x})]\!](e) & = e(p) \text{ si } p \notin D
\end{array}
$$

TAB. 3.1 – *Sémantique dénotationnelle de CC déterministe*

Pour traiter l'opérateur de non-déterminisme, on ajoute naturellement l'équation $[\![D.A+B]\!](e) = [\![D.A]\!](e) \cup [\![D.B]\!](e)$. La dénotation ainsi obtenue reflète bien l'ensemble des stores terminaux d'un agent (voir par exemple [10]), mais on perd cependant la relation précise d'entrée-sortie: en effet $[\![tell(c) + tell(1)]\!] = [\![tell(1)]\!] = \mathcal{C}$, alors que dans le store initial vide ils ont respectivement comme ensemble de stores terminaux $\{c, 1\}$ et $\{1\}$. Il est cependant

important de noter que l'on peut retrouver cette précision grâce à une nouvelle sémantique dénotationnelle [23] qui considère des ensembles d'ensembles de contraintes correspondant intuitivement aux branches de la dérivation:

$$
\begin{array}{ll}
[\![D.tell(c)]\!](e) & = \{\uparrow c\} \\
[\![D.c \rightarrow A]\!](e) & = \{(\uparrow c \cap X) \cup \overline{\uparrow c} \mid X \in [\![D.A]\!](e)\} \\
[\![D.A \parallel B]\!](e) & = \{X \cap Y \mid X \in [\![D.A]\!](e), Y \in [\![D.B]\!](e)\} \\
[\![\exists x A]\!](e) & = \exists_x [\![D.A]\!](e) \\
[\![D.A + B]\!](e) & = [\![D.A]\!](e) \cup [\![D.B]\!](e) \\
[\![D.p(\vec{x})]\!](e) & = \mu\Psi \text{ où } \Psi(f) = [\![D \setminus \{p\}.A]\!](e\{f/p\}) \text{ si } p = A \in D \\
[\![D.p(\vec{x})]\!](e) & = e(p) \text{ si } p \notin D
\end{array}
$$

TAB. 3.2 – *Sémantique dénotationnelle de CC*

$$\mathcal{O}^{term.}(\mathcal{C}, \mathcal{D}.A) = \{d \in \mathcal{C} \mid \exists X \in [\![D.A]\!], d = min(X)\}$$

Le résultat obtenu est d'une nature assez différente de ceux apportés par la sémantique logique: en effet il est ici possible d'avoir une relation *entrée-sortie* précise associant à chaque configuration initiale, l'ensemble des observables lui correspondant, sans avoir de clôture, ni vers le haut (comme c'est le cas avec des sémantiques dénotationnelles plus simples), ni vers le bas (comme nous l'avons vu dans les sections précédentes pour la sémantique logique). Ce résultat est d'ailleurs extensible à une version de CC avec choix gardé "mutuellement exclusif" (voir [23]).

Cette précision des résultats a cependant un prix puisque le fait de s'appuyer sur une sémantique de type dénotationnelle implique l'existence d'un treillis, en général complet, de contraintes, et impose que les agents soient interprétables comme des opérateurs de clôture sur un certain domaine. Ces deux conditions vont complètement à l'encontre d'une possible extension de la sémantique aux agents LCC qui ne sont ni idempotents ni monotones. Quant au système de contraintes, on voudra souvent éviter d'avoir une borne inférieure pour deux contraintes, comme par exemple dans le cas des contraintes de synchronisation.

Enfin, il faut signaler que ce résultat ne porte que sur les stores terminaux et que même si en utilisant le codage de la section 2.4 on peut caractériser aussi les succès (stores terminaux de $A_x^\bullet$ contenant $ok(x)$), on n'a pas autant d'information qu'à la section précédente concernant les suspensions puisqu'on ne connaît pas les agents sur lesquels on suspend, de plus si une contrainte est à la fois un succès et un store terminal, il est impossible de dire si elle est

une suspension stricte ou pas. D'autre part, il semble difficile de généraliser ce type de résultat à des observables comme les *stores accessibles* qui sont eux caractérisables par la sémantique logique.

3.6 Conclusion

La volonté de pouvoir vérifier des propriétés de programmes a donc, comme nous l'avons vu, motivé de nombreux travaux sur la sémantique des langages CC et LCC. Cependant il semble que malgré les récentes avancées, aussi bien du côté logique que du côté dénotationnel, aucune sémantique ne puisse nous apporter à la fois une caractérisation précise et une grande variété d'observables, voire de langages.

Or si l'on se contente d'observables closes vers le haut, quand un agent se réduit vers une configuration de store c dont on veut dire qu'elle n'est pas dangereuse (ne contient pas la contrainte $danger$), on ne pourra pas dire grand chose de l'examen de $\uparrow c$ qui contient $c \wedge danger$.

De même pour des propriétés de type *liveness* et une sémantique close vers le bas. Sans parler dans ce dernier cas des problèmes entraînés par les échecs qui "impliquent tout".

Il y a donc une nécessité de poursuivre notre recherche dans le domaine de la sémantique des langages CC, ce que nous allons faire dans le chapitre suivant. Nous pourrons alors dans les chapitres 5 et 6 utiliser les propriétés sémantiques des langages CC et LCC pour passer concrètement à la preuve de propriétés de programmes.

Chapitre 4

Une nouvelle sémantique logique pour CC

"Your theory is crazy,
but it's not crazy enough to be true."
– Niels Bohr (1885-1962) to a young physicist

4.1 Introduction

Comme nous l'avons vu, la sémantique logique permet de caractériser de nombreuses propriétés observables mais seulement à une clôture près, quant à la sémantique dénotationnelle, elle a l'avantage de donner une caractérisation précise, mais seulement pour certaines observables, les stores terminaux, et via un codage, les succès, de plus il n'y a pas d'espoir de pouvoir étendre cette méthode aux LCC.

Il semble donc naturel de vouloir utiliser la sémantique logique, afin de garder la diversité des observables que l'on peut caractériser, comme les stores accessibles par exemple, et d'espérer rester compatible avec LCC, mais de chercher à affiner cette sémantique pour obtenir l'exactitude, au moins à équivalence logique près, entre sémantique et observables.

Si l'on regarde les problèmes qui entraînent la clôture citée ci-dessus, on peut les regrouper en deux catégories:

- la confusion entre les axiomes non-logiques $\vdash_{\mathcal{C}}$, utilisés pour vérifier l'implication ou non de la garde d'un *ask*, et la relation d'implication de la logique choisie (ILL par exemple) qui reflète plus fidèlement les transitions,

– certaines règles logiques permettant l'affaiblissement non-contrôlé du store ($\exists$, !, etc.).

Pour résoudre la première catégorie de problèmes, l'idée de base de notre travail fut d'essayer de distinguer, au sein même de la théorie, les deux composantes de $ILL_{\mathcal{C},\mathcal{D}}$. Une modalité serait une solution, mais en fait il s'avère qu'il n'est aucunement nécessaire de modifier la logique sur laquelle on s'appuie, la logique linéaire, mais seulement d'enrichir la théorie pour limiter l'action des axiomes non-logiques aux contraintes qui vont être consommées par un *ask*.

La deuxième catégorie de problèmes est un peu différente et il est même difficile d'établir précisément ce que l'on veut que $\exists$ fasse et ce que l'on veut éviter. Nous verrons dans la section suivante comment l'introduction de nouvelles contraintes va résoudre ce problème en permettant de distinguer les $\exists$ introduits par un agent ou une contrainte et ceux découlant de l'affaiblissement d'une contrainte $c(t)$ en $\exists x\ c(x)$. Pour l'exponentiel !, il semble difficile de le traiter correctement, aussi nous verrons comment il est possible de s'en passer et donc de simplifier la logique sur laquelle on s'appuie de ILL a IMALL.

4.2 Sémantique pour CC

4.2.1 Définitions

Soient $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes et $\mathcal{D}$ des déclarations.

Définissons $\mathcal{C}'$ comme le plus petit ensemble tel que:

– pour toute formule atomique a de $\mathcal{C}$, on a $a \in \mathcal{C}'$, et il existe une formule a' dans $\mathcal{C}' \setminus \mathcal{C}$,

– pour toute variable x de $\mathcal{C}$, il existe une formule $loc(x)$ dans $\mathcal{C}' \setminus \mathcal{C}$.

On a donc, intuitivement, dans $\mathcal{C}'$, à la fois $\mathcal{C}$, une copie de $\mathcal{C}$ et des formules $loc(x)$.

Soit $\star$ la traduction de $\mathcal{C}$ dans $\mathcal{C}'$ définie par:

$$(e \wedge f)^\star = e^\star \otimes f^\star \quad (\exists x(e))^\star = \exists x(loc(x) \otimes e^\star) \quad c^\star = c' \text{ si } c \text{ est atomique}$$

Définissons une relation d'implication $\vdash_{\mathcal{C}'}$ sur $\mathcal{C}'$ afin d'en faire un système de contraintes: pour tout $c \vdash_{\mathcal{C}} d$ on a $c^\star \vdash_{\mathcal{C}'} c^\star \otimes d$

On obtient un nouveau système de contraintes, une nouvelle théorie, et on va en tirer parti par une traduction des agents CC en formules, plus raffinée

que celle de la section 3.3.

Définition 4.1 *Les agents CC sont traduits en formules IMALL (logique linéaire intuitionniste sans les exponentiels) de la manière suivante:*

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

Cette traduction s'étend comme d'habitude aux multi-ensembles d'agents et aux configurations:

Si Γ *est le multi-ensemble* $(A_1 \ldots A_n)$*, on définit* $\Gamma^{\Box} = A_1^{\Box} \otimes \cdots \otimes A_n^{\Box}$*. Si* $\Gamma = \emptyset$ *alors* $\Gamma^{\Box} = 1$*.*

La traduction $(X; c; \Gamma)^{\Box}$ *d'une configuration* $(X; c; \Gamma)$ *est la formule* $\exists X (\bigotimes_{x \in X} loc(x) \otimes c^{\star} \otimes \Gamma^{\Box})$*.*

On notera IMALL$(\mathcal{C}', \mathcal{D})$ le système de déduction obtenu en ajoutant à IMALL les axiomes non-logiques suivants:

- $c \vdash d$ pour chaque $c \vdash_{\mathcal{C}'} d$ de $\vdash_{\mathcal{C}'}$,
- $p(\vec{x}) \vdash A^{\Box}$ pour chaque déclaration $p(\vec{x}) = A$ de $\mathcal{D}$.

Théorème 4.2 (Correction) *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *deux configurations CC.*

Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^{\Box} \dashv\vdash_{IMALL(\mathcal{C}', \mathcal{D})} (Y; d; \Delta)^{\Box}$*.*

Si $(X; c; \Gamma) \longrightarrow^{*} (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^{\Box} \vdash_{IMALL(\mathcal{C}', \mathcal{D})} (Y; d; \Delta)^{\Box}$*.*

Preuve.
Par induction sur $\equiv$ et $\longrightarrow$.

- Pour les *composition parallèle*, *α-conversion*, *tell*, *équivalence*, *appels de procédure* et le *choix aveugle* c'est immédiat.
- Pour la *localisation*, $A \otimes \exists x (B \otimes loc(x)) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} \exists x (loc(x) \otimes A \otimes B)$ si $x \notin fv(A)$.
- Pour *ask*, il suffit de remarquer que $c^{\star} \otimes \forall \vec{x}(d \multimap A) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} c^{\star} \otimes d[\vec{t}/\vec{x}] \otimes \forall \vec{x}(d \multimap A) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} c^{\star} \otimes A[\vec{t}/\vec{x}]$ si $c \vdash_{\mathcal{C}} d[\vec{t}/\vec{x}]$, cqfd.

□

4.2.2 Complétude

Bien que ces propriétés de correction soient importantes (voir par exemple les chapitres 5 et 6), elles ne sont pas nouvelles, intéressons nous donc plutôt à la réciproque.

Il s'avère que nous obtenons un résultat de complétude pour l'observation *exacte* des succès.

Notons $\longrightarrow_c^*$ la relation obtenue en ajoutant dans une dérivation $\longrightarrow^*$ la possibilité de débloquer un ask suspendu sur c.

Soient $(X; c; \Gamma)$ une configuration CC, d une contrainte de $\mathcal{C}$, Z un ensemble de variables et ϕ une contrainte de $\mathcal{C}'$ ou un nom de procédure. notons $(X; c; \Gamma), d, Z \Longrightarrow \phi$ pour:

1. si ϕ est de la forme $\bigotimes_{y \in Y} loc(y) \otimes e^\star$ alors $(X; c; \Gamma), d, Z \Longrightarrow \phi$ si $Y \subset Z$ et $(X; c; \Gamma), d, Z \setminus Y \Longrightarrow e^\star$.
2. sinon, si $fv(\phi) \cap Z \neq \emptyset$, alors $(X; c; \Gamma), d, Z \Longrightarrow \phi$ est toujours "vrai".
3. sinon:
   (a) si ϕ est un nom de procédure: $(X; c; \Gamma), d, Z \Longrightarrow \phi$ s'il existe une configuration $(Y; e; \phi)$, telle que $fv(\phi) \cap Y = \emptyset, \exists Y (\bigotimes_{y \in Y} loc(y) \otimes e^\star) \dashv\vdash 1$ et $(X \cup Z; c; \Gamma) \longrightarrow_d^* (Y; e; \phi)$.
   (b) si ϕ est une contrainte de la forme $e^\star$ alors $(X; c; \Gamma), d, Z \Longrightarrow \phi$ s'il existe une configuration *succès* $(Y; e; \emptyset)$, telle que $(Y; e; \emptyset)^\Box \dashv\vdash \phi$ et $(X \cup Z; c; \Gamma) \longrightarrow_d^* (Y; e; \emptyset)$.
   (c) si ϕ est une contrainte de $\mathcal{C}$ alors $(X; c; \Gamma), d, Z \Longrightarrow \phi$ si $d = \phi$ et $(X; c; \Gamma)$ et Z sont vides quand $\phi \neq 1$ ou $(X; c; \Gamma), d$ et Z vides si $\phi = 1$.
   (d) si ϕ est une autre sorte de contrainte alors $(X; c; \Gamma), d, Z \Longrightarrow \phi$ est toujours "vrai".

On notera que si $\kappa \longrightarrow^* \lambda$ et $\lambda, d, Z \Longrightarrow \phi$ alors $\kappa, d, Z \Longrightarrow \phi$ puisque $\Longrightarrow$ est définie par cas sur ϕ en fonction de $\longrightarrow^*$.

Lemme 4.3 *Soient κ et λ deux configurations CC telles que $\kappa^\Box = \lambda^\Box$, d une contrainte de $\mathcal{C}$, X un ensemble de variables et ϕ une contrainte ou un nom de procédure.*

$\kappa, d, X \Longrightarrow \phi$ ssi $\lambda, d, X \Longrightarrow \phi$.

Preuve.
Prouvons le lemme par cas sur la formule $\kappa^\Box = \lambda^\Box$.

- Si $\kappa^\Box = \lambda^\Box$ est atomique, le résultat est évident.

- Si $\kappa^\Box = \lambda^\Box = \forall\vec{x}(c \multimap A^\Box)$, avec c une contrainte et A un agent, alors κ et λ sont nécessairement toutes deux de la forme $(\emptyset; 1; \forall\vec{x}(c \to A))$.
- Si $\kappa^\Box = \lambda^\Box = \exists x(loc(x) \otimes A^\Box)$, alors les deux seules possibilités pour κ et λ sont $(\{x\} \cup Y; c; \Gamma)$ et $(Y_1; 1; \exists x \exists Y_2(tell(c) \parallel \Gamma))$, avec $Y_1 \cup Y_2 = Y$. Comme la deuxième configuration se réduit sur la première et que $\Longrightarrow$ est défini à l'aide de $\longrightarrow^*$ une des implications est triviale. Pour la réciproque il suffit de remarquer que les seules dérivations possibles pour la seconde configuration mènent à $(\{x\} \cup Y; 1; tell(c) \parallel \Gamma)$. On peut alors utiliser la propriété de monotonie 2.11 pour obtenir le résultat.
- Si $\kappa^\Box = \lambda^\Box = A^\Box \otimes B^\Box$, alors les possibilités pour κ et λ sont: $(\emptyset; 1; A \parallel B)$, $(\emptyset; c; B)$ (si $A^\Box = c^\star$, une contrainte, i.e. $A = tell(c)$), $(\emptyset; c; A)$ (si $B^\Box = c^\star$) et $(\emptyset; c \otimes f; \emptyset)$ (si $A^\Box = c^\star$ et $B^\Box = f^\star$, des contraintes). On obtient à nouveau le résultat par dérivation de l'une à l'autre et monotonie.
- Si $\kappa^\Box = \lambda^\Box = A^\Box \,\&\, B^\Box$, alors la seule configuration possible est $(\emptyset; 1; A + B)$, donc $\kappa = \lambda$, cqfd.

□

Lemme 4.4 *Soit $\kappa = (X; c; \Gamma)$ une configuration CC, d une contrainte de $\mathcal{C}$, Y un ensemble de variables et ϕ une contrainte ou un nom de procédure.*
Si $\kappa^\Box, d, \bigotimes_{y \in Y} loc(y) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} \phi$, alors $\kappa, d, Y \Longrightarrow \phi$.

Preuve.

Procédons par induction sur la preuve. Cette induction a un sens puisque l'on peut supposer, sans perte de généralité, que les coupures n'apparaissent qu'au niveau des axiomes non-logiques, et qu'elles sont donc d'une des formes suivantes:

$$\frac{\Gamma, d \vdash p \quad \overline{p \vdash \phi}}{\Gamma, d \vdash \phi} \qquad \frac{\Gamma, d \vdash e^\star \quad \overline{e^\star \vdash e^\star \otimes f}}{\Gamma, d \vdash e^\star \otimes f}$$

$$\frac{\overline{p \vdash \psi} \quad \Gamma, \psi \vdash \phi}{\Gamma, p \vdash \phi} \qquad \frac{\overline{e^\star \vdash e^\star \otimes f} \quad \Gamma, e^\star \otimes f \vdash \phi}{\Gamma, e^\star \vdash \phi}$$

L'application (de bas en haut) d'une de ces règles introduit des séquents dont la formule à droite reste une contrainte ou un nom de procédure.

Il en est de même pour l'introduction à gauche de $\multimap$.

Les formules à gauche du séquent restent quant à elles des sous-formules de traduction d'agents et de contraintes, et sont donc des traductions d'agents et des contraintes.

- Si π est un axiome: on peut utiliser la réflexivité de $\longrightarrow^*$ dans le cas d'un axiome logique, ou la règle *déclarations* pour un axiome $p \vdash q$; le cas d'un axiome $c^\star \vdash c^\star \otimes d$ est trivial.
- π finit par une coupure: nous avons vu que les cas possibles sont ceux énumérés ci-dessus. Considérons donc quelques cas en détail, par exemple:

$$\frac{\Gamma, c, \bigotimes_{x\in X} loc(x) \vdash p \quad \overline{p \vdash \phi}}{\Gamma, c, \bigotimes_{x\in X} loc(x) \vdash \phi}$$

  Par hypothèse d'induction, $(\emptyset; 1; \Gamma)^\Box, c, X \Longrightarrow p$, i.e. il existe une configuration $(Y; d; p)$ telle que $fv(p) \cap Y = \emptyset$ donc $fv(\phi) \cap Y = \emptyset$ (hypothèse sur la formation des déclarations), $(X; 1; \Gamma) \longrightarrow_c^* (Y; d; p)$ et $\exists Y (\bigotimes_{y\in Y} loc(y) \otimes d) \dashv\vdash 1$.
  Si ϕ est un nom de procédure, on a $(X; 1; \Gamma) \longrightarrow_c^* (Y; d; \phi)$ par appel de procédure et $fv(\phi) \cap Y = \emptyset$, cqfd. Si $\phi = e^\star$, alors l'axiome vient d'une déclaration $p = tell(e)$ donc $(X; 1; \Gamma) \longrightarrow_c^* (Y; d; \phi) \longrightarrow^* (Y; d \otimes e; \emptyset)$ et $\exists Y (\bigotimes_{y\in Y} loc(y) \otimes d \otimes e) \dashv\vdash e$ (on a $fv(e) \cap Y = fv(\phi) \cap Y = \emptyset$), cqfd.
  Le seul autre cas intéressant est:

$$\frac{\overline{c^\star \vdash c^\star \otimes d} \quad \Gamma, c^\star \otimes d, \bigotimes_{x\in X} loc(x) \vdash \phi}{\Gamma, c^\star, \bigotimes_{x\in X} loc(x) \vdash \phi}$$

  Par hypothèse d'induction $(\emptyset; 1; \Gamma, tell(c)), d, X \Longrightarrow \phi$, or par définition de $\Longrightarrow$ cela signifie que l'on peut utiliser une fois d pour débloquer un ask. Mais comme $c \vdash_\mathcal{C} d$, c pourrait débloquer ce ask donc $(\emptyset; c; \Gamma), 1, X \Longrightarrow \phi$, cqfd.
  Les deux autres cas se traitent par la réduction d'une configuration à une autre ou par la définition même de $\Longrightarrow$, cas 3.d.
- π finis par une introduction à gauche de 1: on a $(\emptyset; 1; \Gamma, tell(1)) \longrightarrow^* (\emptyset; 1; \Gamma)$. Par hypothèse d'induction, $(\emptyset; 1; \Gamma), c, X \Longrightarrow \phi$ donc $(\emptyset; 1; \Gamma, tell(1)), c, X \Longrightarrow \phi$, cqfd.
- π finit par:

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

Il peut y avoir une ambiguïté quand $A^\Box$ et $B^\Box$ sont des contraintes, cependant grâce au lemme 4.3 on sait que la nature exacte des configurations n'est pas importante.

– π finit par:

$$\frac{\Gamma^\Box, c, \bigotimes_{x\in X} loc(x) \vdash \phi \quad \Delta^\Box, d, \bigotimes_{y\in Y} loc(y) \vdash \psi}{\Gamma^\Box, \Delta^\Box, c, d, \bigotimes_{z\in X\cup Y} loc(z) \vdash \phi \otimes \psi}$$

Si $fv(\phi) \cap X \neq \emptyset$ ou $fv(\psi) \cap Y \neq \emptyset$ alors $fv(\phi \otimes \psi) \cap (X \cup Y) \neq \emptyset$. Si ϕ ou ψ contient un $loc(z)$ alors d'après l'hypothèse d'induction on sait que la variable apparaît dans l'ensemble correspondant (X ou Y) et donc dans leur union, par conséquent on peut considérer que ϕ et ψ sont des contraintes.
Si $\phi = e^\star$ et $\psi = f^\star$, on a $(X; 1; \Gamma) \longrightarrow_c^* (X \cup X'; c'; \emptyset)$ avec $(X \cup X'; c'; \emptyset)^\Box \dashv\vdash e^\star$ et de même pour $(Y; 1; \Delta)$. On obtient alors par monotonie $(X \cup Y; 1; \Gamma, \Delta) \longrightarrow_c^* (X \cup X'' \cup Y; c''; \Delta)$ et $(X \cup X''; c''; \emptyset)^\Box \dashv\vdash e^\star$ puisque nous n'avons fait que des renommages de X' à X'' pour obtenir $X'' \cap (Y \cup fv(d, \Delta)) = \emptyset$. On a d'ailleurs toujours $fv(c'') = fv(\phi)$. Par le même mécanisme on obtient $(X \cup Y; 1; \Gamma, \Delta) \longrightarrow_{c,d}^* (X \cup Y \cup X'' \cup Y''; c'' \wedge d''; \emptyset)$ et $Y'' \cap (X \cup X'' \cup fv(c'')) = \emptyset$. D'où $(X \cup Y \cup X'' \cup Y''; c'' \wedge d''; \emptyset)^\Box = \exists Y \exists X \exists X'' \exists Y''(d'' \otimes c'')$ et comme $Y'' \cap fv(c'') = \emptyset$ cela revient à $\exists Y \exists X \exists X''(\exists Y'' d'' \otimes c'')$. Or les variables libres de $\exists Y'' d''$ viennent de celles de d et Δ et l'on sait que $X'' \cap fv(d, \Delta) = \emptyset$ donc on obtient $\exists Y \exists X(\exists Y'' d'' \otimes \exists X'' c'')$. Par conséquent, si $fv(\phi) \cap Y \neq \emptyset$ (respectivement ψ et Y) on a $fv(\phi \otimes \psi) \cap X \cup Y \neq \emptyset$ et on a donc terminé (par définition de $\Longrightarrow$); sinon on a $fv(c'') \cap Y = \emptyset$ puisque $fv(c'') = fv(\phi)$ et donc $\exists Y \exists Y'' d'' \otimes \exists X \exists X'' c''$ qui est équivalent à $\phi \otimes \psi$, cqfd.
Le cas où ϕ et ψ sont toutes les deux dans $\mathcal{C}$ découle directement de la définition de $\Longrightarrow$, cas 3.c.

– π finit par:

$$\frac{\Gamma^\Box, A^\Box, c, \bigotimes_{x\in X} loc(x) \vdash \phi}{\Gamma^\Box, A^\Box \,\&\, B^\Box, c, \bigotimes_{x\in X} loc(x) \vdash \phi}$$

Par hypothèse d'induction, $(\emptyset; 1; A, \Gamma), c, X \Longrightarrow \phi$, or $(\emptyset; 1; A + B, \Gamma) \longrightarrow^* (\emptyset; 1; A, \Gamma)$, et X ne change pas, donc $(\emptyset; 1; A + B, \Gamma), c, X \Longrightarrow \phi$.

– π finit par une introduction à droite de $\exists$. Le seul cas non-trivial est quand ϕ est une contrainte $e^\star = \exists x(loc(x) \otimes c^\star)$ (sinon on peut utiliser le cas 3.d. de la définition de $\Longrightarrow$).

$$\frac{\Gamma^\Box, d, \bigotimes_{y\in Y} loc(y) \vdash loc(t) \otimes c^\star[t/x]}{\Gamma^\Box, d, \bigotimes_{y\in Y} loc(y) \vdash \exists x(loc(x) \otimes c^\star)}$$

Par hypothèse d'induction on doit avoir $t \in Y$ et $(\emptyset; 1; \Gamma), d, Y \setminus \{t\} \Longrightarrow c^\star[t/x]$, i.e. il existe un succès $(Z; e; \emptyset)$ tel que $(Y \setminus \{t\}; 1; \Gamma) \longrightarrow_d^* (Z; e; \emptyset)$ et $(Z; e; \emptyset)^\square \dashv\vdash c^\star[t/x]$. Si on ajoute t à $Y \setminus \{t\}$ on ne fait que bloquer une dérivation où apparaîtrait un agent $\exists t A$, mais on sait que l'on peut renommer t par α-conversion et obtenir une nouvelle dérivation avec un résultat équivalent. Donc $(Y; 1; \Gamma) \longrightarrow_d^* (Z' \cup \{t\}; e'; \emptyset)$ et on a $(Z' \cup \{t\}; e'; \emptyset)^\square \dashv\vdash \exists t(loc(t) \otimes c^\star[t/x]) \dashv\vdash \exists x(loc(x) \otimes c^\star)$, cqfd.

– π finit par:

$$\frac{\Gamma^\square, c, \bigotimes_{y \in Y} loc(y), A^\square \otimes loc(x) \vdash \phi}{\Gamma^\square, c, \bigotimes_{y \in Y} loc(y), \exists x(A^\square \otimes loc(x)) \vdash \phi}\ x \notin fv(\Gamma, c, \phi) \cup Y$$

Par hypothèse d'induction, $(\emptyset; 1; A, \Gamma), c, Y \cup \{x\} \Longrightarrow \phi$. Comme $x \notin fv(\phi)$ on sait que ϕ n'est pas de la forme $loc(x) \otimes \psi$, de plus si $fv(\phi) \cap (Y \cup \{x\}) \neq \emptyset$ alors $fv(\phi) \cap Y \neq \emptyset$ et on a déjà terminé, sinon on a $(Y \cup \{x\}; 1; A, \Gamma) \longrightarrow_c^* \kappa$ pour un κ (avec les propriétés de $\Longrightarrow$). Mais comme $x \notin fv(\Gamma, c, \phi) \cup Y$, $(Y; 1; \exists x A, \Gamma) \longrightarrow (Y \cup \{x\}; 1; A, \Gamma)$, donc $(Y; 1; \exists x A, \Gamma) \longrightarrow_c^* \kappa$, cqfd.

– π finit par:

$$\frac{\Gamma^\square, A^\square, d \vdash \phi \quad \Delta^\square, e \vdash c}{\Gamma^\square, \Delta^\square, c \multimap A^\square, d, e \vdash \phi}$$

Par hypothèse d'induction, $(\emptyset; 1; \Delta), e \Longrightarrow c$, i.e. $e = c$ et Δ est vide. Or $(\emptyset; 1; c \to A, \Gamma) \longrightarrow_c^* (\emptyset; 1; A, \Gamma)$ par définition de $\longrightarrow_c$. D'où $(\emptyset; 1; c \to A, \Delta, \Gamma), d, e \Longrightarrow \phi$.

– π finit par:

$$\frac{\Gamma^\square, A^\square[t/x], d \vdash \phi}{\Gamma^\square, \forall x A^\square, d \vdash \phi}$$

Cette situation ne se présente bien sur qu'avec un *ask* pour A, on vérifie alors aisément que si un $c(t) \to B(t)$ permet une dérivation où ce *ask* disparaît (puisqu'il ne peut apparaître dans ϕ), $\forall x(c(x) \to B(x))$ permet la même.

□

Notons $=_{\dashv\vdash}$ l'égalité modulo $\dashv\vdash$. On peut désormais énoncer le théorème de complétude qui étend et précise celui du chapitre 3.

Théorème 4.5 (Observation exacte des succès) *Soit A un agent CC. Définissons $\mathcal{LL}_\square^{succès}(\mathcal{C}, \mathcal{D}.A) = \{c \in \mathcal{C} | A^\square \vdash_{IMALL(\mathcal{C}', \mathcal{D})} c^\star\}$. On a*

$$\mathcal{LL}^{succès}_{\Box}(\mathcal{C}, \mathcal{D}.A) =_{\dashv\vdash} \mathcal{O}^{succès}(\mathcal{C}, \mathcal{D}.A).$$

Preuve.
Il suffit d'appliquer le lemme précédent à la configuration $(\emptyset; 1; A)$ et à $\phi = c^\star$ puis de se référer à la définition de $\Longrightarrow$ (cas 3.b.). □

Remarque.
On doit tout de même se contenter de $=_{\dashv\vdash}$, c'est à dire de l'équivalence logique entre les contraintes observées et celles de la sémantique logique, car on ne peut pas faire disparaître les quantificateurs existentiels, même inutiles, dans la sémantique opérationnelle, alors que cela peut se produire dans certains cas pour des contraintes (par exemple $(\emptyset; 1; \exists x\ p(y)) \longrightarrow (\{x\}; 1; p(y))$ dont le store est bien équivalent à 1, mais on ne peut pas se débarrasser du x).

4.3 LCC

Le cas des langages LCC n'est pas beaucoup plus compliqué à traiter, en effet nous nous sommes déjà appuyés sur une sémantique en logique linéaire, il est donc assez naturel d'y traduire les agents LCC. Il faudra cependant prendre garde à la définition de notre nouvelle théorie ($\vdash_{\mathcal{C}'}$) afin de gérer correctement la consommation de contraintes. De plus nous devons éviter la présence de l'exponentiel ! qui permet d'*affaiblir* le store sans contrôle. On doit donc se restreindre à un fragment de LCC où toutes les contraintes apparaîssent sous une forme purement linéaire, cependant comme la section précédente l'a montré, cela n'enlève pas tant de pouvoir expressif que cela puisque les contraintes "classiques" peuvent être codées en les répétant à droite du symbole $\vdash$.

4.3.1 Définitions

Soient $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes linéaire (sans "!") et $\mathcal{D}$ des déclarations. $\mathcal{C}'$ est défini comme dans la section 4.2.1.

Soit $\star\star$ la traduction de $\mathcal{C}$ dans $\mathcal{C}'$ définie par:

$$(e \otimes f)^{\star\star} = e^{\star\star} \otimes f^{\star\star} \quad (\exists x(e))^{\star\star} = \exists x(loc(x) \otimes e^{\star\star}) \quad c^{\star\star} = c' \text{ si } c \text{ est atomique}$$

Définissons une relation d'implication $\vdash_{\mathcal{C}'}$ sur $\mathcal{C}'$ afin d'en faire un système de contraintes: pour tout $c \vdash_{\mathcal{C}} \bigotimes_{i\in I} d_i$ et tout $J \subsetneq I, J \neq \emptyset$ posons $c^{\star\star} \vdash_{\mathcal{C}'} \bigotimes_{i\in I\setminus J} d_i \ \otimes \bigotimes_{j\in J} d_i^{\star\star}$.

On obtient à nouveau une nouvelle théorie, que l'on va utiliser pour traduire les agents LCC exactement comme dans la section 4.2.1, sauf pour $tell(c)$ où l'on utilise bien sur désormais $c^{\star\star}$. On notera aussi cette traduction $\Box$ par abus de notation.

IMALL$(\mathcal{C}', \mathcal{D})$ est défini comme précédemment.

Théorème 4.6 (Correction) *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *deux configurations LCC.*

Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^{\Box} \dashv\vdash_{IMALL(\mathcal{C}', \mathcal{D})} (Y; d; \Delta)^{\Box}$.

Si $(X; c; \Gamma) \longrightarrow^*_{LCC} (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^{\Box} \vdash_{IMALL(\mathcal{C}', \mathcal{D})} (Y; d; \Delta)^{\Box}$.

Preuve.
Par induction sur $\equiv$ et $\longrightarrow_{LCC}$, la preuve est la même que pour CC sauf pour le *ask* où l'on vérifie que l'on a bien $c^{\star\star} \otimes \forall \vec{x}(d \multimap A) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} e^{\star\star} \otimes d[\vec{t}/\vec{x}] \otimes \forall \vec{x}(d \multimap A) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} e^{\star\star} \otimes d[\vec{t}/\vec{x}] \otimes (d \multimap A)[\vec{t}/\vec{x}] \vdash_{IMALL(\mathcal{C}', \mathcal{D})} e^{\star\star} \otimes A[\vec{t}/\vec{x}]$ si $c \vdash_{\mathcal{C}} d[\vec{t}/\vec{x}] \otimes e$. Comme on peut toujours prendre d ou e égal à 1 donc les restrictions $J \neq I$ et $J \neq \emptyset$ de la définition de $\vdash_{\mathcal{C}'}$ ne posent pas de problème. $\Box$

4.3.2 Complétude

Pour LCC aussi nous obtenons un résultat de complétude pour l'observation *exacte* des succès, et ceci en changeant très peu la preuve de la section précédente.

$\Longrightarrow$ est définie de façon identique, si ce n'est que l'on remplacera $(X; c; \Gamma) \longrightarrow^*_d (Y; e; \Delta)$ par $(X; c; \Gamma, tell(d)) \longrightarrow^*_{LCC} (Y; e; \Delta)$ qui a intuitivement la même signification.

Le lemme 4.3 reste valable.

Lemme 4.7 *Soit* $\kappa = (X; c; \Gamma)$ *une configuration LCC,* d *une contrainte de* $\mathcal{C}$, Y *un ensemble de variables et* ϕ *une contrainte ou un nom de procédure.*

Si $\kappa^{\Box}, d, \bigotimes_{y \in Y} loc(y) \vdash_{IMALL(\mathcal{C}', \mathcal{D})} \phi$, *alors* $\kappa, d, Y \Longrightarrow \phi$.

Preuve.
La preuve suivant exactement la même structure que dans la section précédente, nous ne nous intéresserons qu'aux cas qui ont changé:

- π finit par une coupure sur un axiome non logique:

$$\frac{\overline{c^{\star\star} \vdash d^{\star\star} \otimes e} \quad \Gamma, d^{\star\star} \otimes e, \bigotimes_{x \in X} loc(x) \vdash \phi}{\Gamma, c^{\star\star}, \bigotimes_{x \in X} loc(x) \vdash \phi}$$

Par hypothèse d'induction $(\emptyset; d; \Gamma), e, X \Longrightarrow \phi$, et on a $c \vdash_{\mathcal{C}} d \otimes e$. D'après la définition de $\Longrightarrow$, si on n'est pas dans un cas trivial on a donc $(X; d; \Gamma, tell(e)) \longrightarrow^*_{LCC} \kappa$ avec certaines propriétés pour κ, Il suffit donc d'utiliser la propriété de monotonie (toujours valable dans LCC) pour obtenir $(X; c; \Gamma) \longrightarrow^*_{LCC} \kappa$ et donc $(\emptyset; c; \Gamma), 1, X \Longrightarrow \phi$, cqfd.

– π finit par:

$$\frac{\Gamma^{\Box}, A^{\Box}, d \vdash \phi \quad \Delta^{\Box}, e \vdash c}{\Gamma^{\Box}, \Delta^{\Box}, c \multimap A^{\Box}, d, e \vdash \phi}$$

Par hypothèse d'induction, $(\emptyset; 1; \Delta), e \Longrightarrow c$, i.e. $e = c$ et Δ est vide. Or $(\emptyset; 1; c \to A, \Gamma, tell(c)) \longrightarrow (\emptyset; 1; A, \Gamma)$. D'où $(\emptyset; 1; c \to A, \Delta, \Gamma), d, e \Longrightarrow \phi$.

□

Le théorème de complétude précise à nouveau celui du chapitre 3.

Théorème 4.8 (Observation exacte des succès) *Soit A un agent LCC. On a*

$$\mathcal{LL}^{succès}_{\Box}(\mathcal{C}, \mathcal{D}.A) =_{\dashv\vdash} \mathcal{O}^{succès}_{LCC}(\mathcal{C}, \mathcal{D}.A).$$

Preuve.
Il suffit d'appliquer le lemme précédent à la configuration $(\emptyset; 1; A)$ et à $\phi = c^{\star\star}$ puis de se référer à la définition de $\Longrightarrow$ (cas 3.b.). □

4.4 Stores

On sait depuis longtemps caractériser les *stores* des agents CC en utilisant la logique intuitionniste (voir chapitre 3), on pourrait donc penser que cette observable est facile d'accès et que la méthode des sections précédentes va permettre directement de gagner ici aussi en précision, que ce soit pour CC ou LCC.

Cependant il faut noter que lorsque l'on se contente des *conséquences* des stores accessibles on dissimule une difficulté qui consiste à différencier une contrainte dans le store et l'agent qui va l'y mettre. Ainsi on a jusqu'à présent toujours interprété de la même façon $(\emptyset; 1; tell(c), tell(d)), (\emptyset; c; tell(d))$ et $(\emptyset; c \otimes d; \emptyset)$. Cela nous à entraîné à confondre les agents $tell(c \otimes d)$ et $tell(c) \parallel tell(d)$ qui si ils ont bien les mêmes succès, suspensions et *conséquences de*

stores accessibles n'ont pourtant pas les mêmes *stores*: le deuxième agent pouvant bien sur atteindre les stores c et d qui sont inaccessibles pour le premier.

On va donc à nouveau avoir à raffiner notre sémantique en modifiant quelque peu la théorie et la traduction des agents en formules. Nous traiterons ici directement le cas de LCC, celui de CC en étant un cas particulier.

4.4.1 Définitions

Soient $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes et $\mathcal{D}$ des déclarations.

Définissons $\mathcal{C}''$ comme le plus petit ensemble tel que:

- pour toute formule c de $\mathcal{C}'$ (défini comme auparavant), $c \in \mathcal{C}''$,
- pour toute contrainte c (éventuellement non-atomique) de $\mathcal{C}$, il existe une contrainte $c'' \in \mathcal{C}'' \setminus \mathcal{C}'$.

On a donc, intuitivement, dans $\mathcal{C}''$, à la fois $\mathcal{C}'$ et une copie de toutes les contraintes de $\mathcal{C}$.

La traduction $\star$ de $\mathcal{C}$ dans $\mathcal{C}''$ est définie comme précédemment.

Définissons une relation d'implication $\vdash_{\mathcal{C}''}$ sur $\mathcal{C}''$ afin d'en faire un système de contraintes linéaire: pour tout $c \vdash_{\mathcal{C}} \bigotimes_{i\in I} d_i$ et tout $J \subsetneq I, J \neq \emptyset$ posons comme dans la section précédente $c^\star \vdash_{\mathcal{C}'} \bigotimes_{i\in I\setminus J} d_i \ \otimes \bigotimes_{j\in J} d_i^\star$. De plus on a pour toute contrainte c de $\mathcal{C}$, $c'' \vdash_{\mathcal{C}''} c^\star$.

Définition 4.9 *Les agents LCC sont traduits en formules IMALL de la manière suivante:*

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

Cette traduction s'étend comme d'habitude aux multi-ensembles d'agents.

La traduction $(X; c; \Gamma)^\Box$ *d'une configuration* $(X; c; \Gamma)$ *est la formule* $\exists X(\bigotimes_{x\in X} loc(x) \otimes c^\star \otimes \Gamma^\Box)$.

Remarque.
On a donc gardé une traduction très similaire à celle des sections précédentes si ce n'est que $tell(c)^\Box = c''$ implique désormais *strictement* $(\emptyset; c; \emptyset)^\Box = c^\star$.

IMALL$(\mathcal{C}'', \mathcal{D})$ reste défini comme précédemment.

Théorème 4.10 (Correction) *Soient* $(X; c; \Gamma)$ *et* $(Y; d; \Delta)$ *deux configurations LCC.*

Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^\Box \dashv\vdash_{IMALL(\mathcal{C}'',\mathcal{D})} (Y; d; \Delta)^\Box$.

Si $(X; c; \Gamma) \longrightarrow^*_{LCC} (Y; d; \Delta)$ *alors* $(X; c; \Gamma)^\Box \vdash_{IMALL(\mathcal{C}'',\mathcal{D})} (Y; d; \Delta)^\Box$.

Preuve.
Par induction sur $\equiv$ et $\longrightarrow_{LCC}$, la preuve est la même que pour LCC sauf pour le *tell* où l'on vérifie que l'on a bien $c'' \vdash c^\star$ cqfd. □

4.4.2 Complétude

On obtient l'analogue du théorème 4.8 pour l'observation des stores:

Théorème 4.11 (Observation exacte des stores) *Soit A un agent LCC. Définissons $\mathcal{LL}^{store}_{\Box}(\mathcal{C}, \mathcal{D}.A) = \{\exists X c \in \mathcal{C} | \exists \Gamma, A^\Box \vdash_{IMALL(\mathcal{C}'', \mathcal{D})} (X; c; \Gamma)^\Box\}$. On a*

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

Preuve.
Concernant l'inclusion correspondant à la complétude, on remarque que si $A^\Box \vdash_{IMALL(\mathcal{C}'', \mathcal{D})} (X; c; \Gamma)^\Box$ alors la preuve commence (de bas en haut) nécessairement par des éliminations à gauche, correspondant exactement aux réductions de A, puis une élimination des $\exists$ des deux côtés, enfin une séparation du membre gauche en deux sous-configurations, κ et κ' telles que $\kappa \vdash c^\star$ et $\kappa' \vdash \Gamma^\Box$. On peut alors appliquer le théorème 4.8 à κ puis remarquer que le store de κ' est nécessairement vide puisque sinon une contrainte de la forme $d^\star$ apparaîtrait dans le membre de droite (puisque on a imposé $J \neq \emptyset$ dans les axiomes non-logiques et que la coupure est la seule règle qui permettrait de faire "disparaître" $d^\star$). κ' est donc juste un agent et par monotonie, l'ajouter à κ permettra les mêmes dérivations, pour obtenir une configuration dont le store est précisément $\exists X c$. □

4.5 Suspensions

Dans [30], Ruet montre qu'il est possible de caractériser les suspensions des agents LCC en utilisant la logique non-commutative (NL, voir l'appendice C), sous réserve de n'utiliser qu'un système de contraintes "trivial". En effet le fait de confondre un store et ses conséquences crée de fausses suspensions. Comme nous pouvons désormais éviter ces affaiblissements incontrôlés du store, il est naturel d'essayer de généraliser le théorème de complétude à l'observation des suspensions.

Nous définissons $\mathcal{C}'$ et $\vdash_{\mathcal{C}'}$ comme auparavant (section 4.3) mais en y ajoutant une nouvelle contrainte $\diamond$ (comme dans [30]) et un nouveau symbole

de relation *test*, d'arité un, destiné à éviter les suspensions dues au fait que $\forall x(c(x) \multimap A(x)) \vdash c(t) \multimap A(t)$; ce cas ne se présentait pas auparavant puisque ϕ ne pouvait jamais être une suspension. On aura de plus pour tout x, $1 \vdash_{\mathcal{C}'} test(x)$.

La traduction des agents LCC en formules reste identique, sauf pour le *ask* qui utilise une variante de la traduction de Ruet: $(\forall x(c \to A))^{\Box} = \forall x((c \otimes test(x)) \multimap (\diamond \otimes A^{\Box}))$ On ajoutera au besoin un $\forall x$ (inutile) aux agents qui n'en ont pas.

La correction reste évidente, aux $\vec{\diamond}$ résidus près, puisque les axiomes non-logiques de *test* ont été ajoutés exprès.

Théorème 4.12 (Observation exacte des suspensions) *Soit A un agent LCC et c une contrainte linéaire. Si $A^{\Box} \vdash_{IMANL(\mathcal{C}',\mathcal{D})} \vec{\diamond} \otimes c^{\star} \otimes \forall x_1((d_1 \otimes test(x_1)) \multimap (\diamond \otimes A_1)) \otimes \cdots \otimes \forall x_n((d_n \otimes test(x_n)) \multimap (\diamond \otimes A_n))$, avec pour aucun $d_i, c \vdash_{\mathcal{C}} d_i[t/x_i] \otimes e_i$, alors A suspend avec le store c sur les contraintes d_i.*

Preuve.
La preuve est une adaptation de celle du théorème 4.8: à la définition de $\Longrightarrow$ on ajoute le cas où ϕ est une suspension qui est du même type que quand ϕ est un $c^{\star}$ (cas 3.b.). Ensuite, on procède à la même induction avec simplement à considérer en plus la possibilité qu'une preuve se termine par la règle $\multimap$ à droite, mais on peut alors utiliser le même raisonnement que Ruet pour remarquer que vu la structure des séquents NL, cette règle est nécessairement suivie d'une règle $\multimap$ à gauche et d'un axiome, on peut donc les regrouper en

$$\frac{A \vdash B}{c \multimap A \vdash c \multimap B}$$

et l'étape d'induction est alors évidente.

Il faut aussi noter que le cas de la règle du $\forall$ à gauche change un petit peu: en effet $c[t/x] \otimes test(t) \multimap \diamond \otimes A[t/x]$ n'est pas la traduction correcte d'un agent; mais on peut remarquer aisément que la seule manière dont cette formule peut impliquer une traduction correcte d'agent est si le *ask* est débloqué, on aurait alors effectivement pu atteindre la même configuration à partir de $\forall x(c \otimes test(c) \multimap \diamond \otimes A)$. □

4.6 Conclusion

Nous avons donc obtenu une sémantique *précise* pour toutes les observables que nous considérions, et ce sans compliquer la logique sous-jacente,

mais seulement en enrichissant la théorie. La seule restriction a été la disparition du !, mais comme nous l'avons vu il peut la plupart du temps être remplacé par une bonne utilisation du système de contraintes. De plus cette disparition nous a même permis de nous restreindre a IMALL (ou IMANL pour le cas non-commutatif) ce qui simplifie un peu l'utilisation de la sémantique obtenue.

Les résultats obtenus sont donc des raffinements de résultats antérieurs où l'on a enfin réussi à combiner les avantages des sémantiques logiques et dénotationnelles définies dans le chapitre précédent.

Nous allons désormais voir comment utiliser la sémantique logique de CC et LCC pour prouver des propriétés de programmes, nous constaterons d'ailleurs à cette occasion, qu'il est bien agréable de pouvoir se passer des exponentiels.

Chapitre 5

Sémantique des phases

"**LOGIC**, n. The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding . . . "
– Ambrose Bierce (1842-1914), "The Devil's Dictionary", 1911

5.1 Introduction

L'outil que nous allons utiliser ici est la sémantique des phases de la logique linéaire. C'est la sémantique naturelle de la prouvabilité pour LL [14] et comme nous avons vus que nos résultats de sémantique logique établissent un lien entre existence d'une dérivation CC ou LCC et prouvabilité, c'est un outil suffisant pour ce qui nous intéresse. Avec une sémantique des preuves on pourrait sans doute établir des propriétés plus fines sur le déroulement même d'une dérivation (et non pas son existence), mais cela dépasse largement le cadre de notre étude.

La sémantique des phases est correcte et complète vis à vis de LL, mais nous verrons que sa structure rend la complétude peu utile: on sait qu'il existe un modèle quand il existe une preuve, mais ce modèle peut parfaitement être le modèle syntaxique (voir [14]) et ne rien nous dire d'autre que "la formule est prouvable dans LL". Par contre, la correction, bien que considérée souvent comme élémentaire, va nous fournir la possibilité de choisir un modèle où nous allons nous abstraire autant que l'on veut des formules (et donc du programme) tout en assurant que ce modèle reflétera les dérivations des programmes LCC. Nous en tirerons parti, principalement, pour prouver des propriétés de sûreté, dans la section 5.3.

5.2 Sémantique des phases pour ILL

Bien que nous n'ayons besoin que d'un fragment réduit d'ILL, il est plus simple de rappeler la définition de la sémantique des phases pour tout ILL (le calcul des séquents complet est donné dans l'appendice A), en suivant la présentation d'Okada [26], et de l'étendre aux constantes ($\mathbf{1}$, $\mathbf{0}$, $\top$).

Définition 5.1 *Un* espace de phases $\mathbf{P} = (P, \cdot, 1, \mathcal{F})$ *est une structure telle que* $(P, \cdot, 1)$ *est un monoïde commutatif, et* $\mathcal{F}$*, appelé* ensemble des faits*, est un sous-ensemble des parties de* P *vérifiant:*

- $\mathcal{F}$ *est clos par intersection arbitraire,*
- *pour tout* $A \subset P$*, pour tout* $F \in \mathcal{F}$*,* $A \multimap F = \{x \in P : \forall a \in A, a \cdot x \in F\}$ *est un fait.*

Ces faits correspondent intuitivement (et en fait, nous le verrons dans la section 5.3, plus que cela) aux formules ILL et donc aux agents CC.

On peut remarquer que les faits sont clos par *implication linéaire* $\multimap$. Voici quelques faits: le plus grand $\top = P$, le plus petit $\mathbf{0}$, et $\mathbf{1} = \bigcap\{F \in \mathcal{F} : 1 \in F\}$.

Un *fait paramétrique* A est une fonction totale de V dans $\mathcal{F}$, assignant à toute variable x un fait $A(x)$. Tout fait peut ainsi être vu comme un fait paramétrique constant, et toute opération définie sur les faits s'étend naturellement aux faits paramétriques: $(A \star B)(x) = A(x) \star B(x)$.

Soient A et B deux faits (paramétriques), on définit les faits suivants:

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

Définition 5.2 *Un* espace de phase enrichi *est un espace de phase* $(P, \cdot, 1, \mathcal{F})$ *avec un sous-ensemble* O *de* $\mathcal{F}$*, dont les éléments sont appelés* faits ouverts*, tel que:*

- O *est clos par* $\oplus$ *arbitraire (en particulier il existe un plus grand fait ouvert),*

– $\mathbf{1}$ *est le plus grand fait ouvert,*
– O *est clos par* $\otimes$ *fini,*
– $\otimes$ *est idempotent sur* O *(si* $A \in O$ *alors* $A \otimes A = A$*).*

$!A$ *est alors défini comme le plus grand fait ouvert inclus dans* A*.*

Ayant donné aux faits des opérateurs correspondants à ceux de ILL, nous pouvons traduire les formules en faits:

Définition 5.3 *Une* valuation η *est une fonction des formules atomiques dans les faits, telle que* $\eta(\top) = \top$, $\mathbf{1} = \eta(\mathbf{1})$ *et que* $1 \in \mathbf{1}$, $\eta(\mathbf{0}) = \mathbf{0}$.

On définit inductivement l'interprétation $\eta(A)$ (resp. $\eta(\Gamma)$) d'une formule A (resp. d'un antécédent Γ) comme suit:

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

Les séquents peuvent alors être interprétés ainsi:

$$\eta(\Gamma \vdash A) = \eta(\Gamma) \multimap \eta(A)$$

Ceci nous amène à définir une notion de validité:

Définition 5.4 (Validité) *On définit:*
$\mathbf{P}$, $\eta \models (\Gamma \vdash A)$ *ssi* $1 \in \eta(\Gamma \vdash A)$, *soit* $\eta(\Gamma) \subset \eta(A)$,
$\mathbf{P} \models (\Gamma \vdash A)$ *ssi pour toute valuation* η*:* $\mathbf{P}$, $\eta \models (\Gamma \vdash A)$,
$\models (\Gamma \vdash A)$ *ssi pour tout espace de phases* $\mathbf{P}$*:* $\mathbf{P} \models (\Gamma \vdash A)$.

Cette sémantique des formules ILL jouit de certaines propriétés:

Théorème 5.5 (Correction [14, 26])

$$\Gamma \vdash_{ILL} A \text{ implique } \models (\Gamma \vdash A).$$

Théorème 5.6 (Complétude [14, 26])

$$\models (\Gamma \vdash A) \textit{ implique } \Gamma \vdash_{ILL} A.$$

Cette dernière propriété ne nous servira pas directement à prouver des propriétés de programmes, mais plutôt à se convaincre de l'existence d'une preuve sémantique de ces propriétés comme décrit dans la section 5.5.

5.3 Preuves sémantiques

Nous allons maintenant exposer la méthode de preuve qui découle de la juxtaposition des résultats précédents. Basée sur les théorèmes de correction de la sémantique des phases et de la traduction LCC-ILL, cette méthode nécessite soit de traduire les programmes CC en LCC comme indiqué en fin de section 2.5.2, soit de coder directement les algorithmes et protocoles en LCC.

Cette deuxième alternative se révélera en fait très puissante, comme le montrent les exemples de la section 5.4. On y verra aussi que le lien entre logique linéaire et CC linéaires est bien plus fort que celui entre logique intuitionniste et CC, puisque, bien qu'on dispose aussi dans ce dernier cas de résultats de correction pour les traductions, il se révèle impossible de faire la plupart des preuves. De fait ce sont les résultats de complétude partielle de la première traduction qui nous assurent de l'existence de preuves sémantiques comme expliqué dans la section 5.5.

Notre méthode de preuves sémantiques découle directement de la proposition suivante, corollaire des deux théorèmes de correction vus précédemment:

Proposition 5.7 (Preuves sémantiques) *Pour prouver une propriété de sûreté de la forme:* $(\vec{x}; c; \Gamma) \not\longrightarrow (\vec{y}; d; \Delta)$*, il suffit de montrer:*
$\exists$ *un espace de phases* $\mathbf{P}$*, une valuation* η *, et un élément* $a \in \eta((\vec{x}; c; \Gamma)^\dagger)$ *tels que* $a \notin \eta((\vec{y}; d; \Delta)^\dagger)$.

Preuve.

Le théorème 5.5 de correction de la sémantique des phases vis à vis de ILL s'énonce comme suit:

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

Ceci peut aisément être étendu à $ILL_{\mathcal{C},\mathcal{D}}$ en imposant à toute valuation η de satisfaire les inclusions qui proviennent des axiomes non-logiques (l'axiome $c \vdash d$ imposant $\eta(c) \subset \eta(d)$).

La contraposée de cette extension nous donne:

$$\exists \mathbf{P}, \eta, \text{ tels que } \mathbf{P}, \eta \not\models (\Gamma \vdash A) \text{ implique } \Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A.$$

C'est à dire:

$$\exists \mathbf{P}, \eta, \text{ tels que } \eta(\Gamma) \not\subset \eta(A) \text{ implique } \Gamma \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} A.$$

Or la contraposée du théorème 3.5 de correction de la traduction de LCC dans $ILL_{\mathcal{C},\mathcal{D}}$ nous donne:

$$(\vec{x}; c; \Gamma)^\dagger \not\vdash_{ILL_{\mathcal{C},\mathcal{D}}} (\vec{y}; d; \Delta)^\dagger \text{ implique } (\vec{x}; c; \Gamma) \not\longrightarrow (\vec{y}; d; \Delta)$$

La juxtaposition de ces deux théorèmes nous donne la proposition. □

Ce corollaire permet de réduire la preuve de propriétés de sûreté d'un programme CC, un problème de non-existence de dérivation, à un problème d'existence, celle d'une structure de phases, d'une interprétation et d'un contre-exemple pour l'inclusion ci-dessus.

On aboutit donc à une méthode en quatre points:

1. Formuler la propriété de sûreté sous la forme: $(\vec{x}; c; \Gamma) \not\longrightarrow (\vec{y}; d; \Delta)$,
2. Exhiber une structure de phases,
3. Exhiber une valuation,
4. Prouver l'existence d'un contre-exemple.

Remarque.
On pourrait se contenter de preuves d'existence pour les points 2 et 3 ci-dessus, cependant on perdrait alors un des bénéfices de cette méthode: la possibilité de travailler dans le cadre simple, d'espaces de phases bien choisis et bien connus. Le dernier point serait aussi rendu plus difficile. On verra dans le chapitre 6 que cette méthode peut en partie s'automatiser.

5.4 Exemples

Voici quelques exemples d'application de cette méthode à des protocoles et des algorithmes standards, pour montrer des propriétés de sûreté classiques comme l'absence de deadlock par exemple.

5.4.1 Exemple 1 - Le dîner des philosophes

Nous avons vu dans la section 2.5.3 que le problème classique du dîner des philosophes pouvait facilement être codé dans LCC, essayons maintenant de prouver des propriétés sur ce programme.

La première propriété qu'on peut vouloir prouver quand on vient d'encoder en LCC un algorithme est sa fidélité vis à vis de l'algorithme d'origine. En l'occurrence on va tenter de démontrer que ce codage est sûr dans le sens où une fourchette ne peut être utilisée que par un philosophe à la fois, c'est à dire que deux philosophes voisins ne peuvent pas manger en même temps, quelque soit le nombre de philosophes.

1. Reformulons la propriété.
Nous devons tout d'abord exprimer notre propriété sous la forme de la non-existence d'une dérivation, soit:

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

D'après le corollaire 5.7 il suffit donc de montrer:

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

C'est à dire, en utilisant la constante $\top$ dont le sens est intuitivement "quelque chose":

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

2. Espace de phases.
Considérons la structure $\mathbf{P}$ suivante:

- $\mathbb{N}$ (avec son produit et son élément neutre habituels) est le monoïde,
- $\mathcal{F} = \mathcal{P}(\mathbb{N})$,
- $O = \{\emptyset, \{1\}\}$.

C'est clairement une structure de phases. En effet $\mathcal{F}$ est clos par intersection et implication et O clos par $\oplus$ et $\otimes$, ce dernier étant bien idempotent sur O.

3. Valuation.
Nous devons maintenant définir une valuation η sur `N=M`, `N≠M`, `fork(I)`, `eat(I,N)`, `philosophe(I,N)`, `recphilo(M,P)` et `init(N)`. Il ne faut pas oublier de vérifier que les conditions provenant des déclarations (axiomes non-logiques qui se traduisent par des inclusions obligatoires) de `philosophe(I,N)`, `recphilo(M,P)` et `init(N)` sont satisfaites.

Définissons η ainsi:

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

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

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

Où les indices (i, m, n, p) sont les interprétations canoniques des variables entières correspondantes, les f_i et les p_i sont des entiers premiers distincts, $e_{i,n}$, $x_{m,p}$ et $y_{m,p}$ sont définis comme suit:

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

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

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

Les conditions venant des déclarations sont:
– $\forall i, \{p_i\} \subset E_{1i} = \eta$(définition de `philosophe(I,N)`)
– $\forall m, \forall p, \{x_{m,p} \cdot y_{m,p}\} \subset F_{m,p}$ où $F_{m,p} = \eta$(définition de `recphilo(M,P)`),
– $\forall n, \{x_{1,n} \cdot y_{1,n}\} \subset \eta$(définition de `init(N)`).

On notera que la troisième condition est une conséquence de la deuxième, puisqu'un agent de la forme $c \to d \to e \to A$ est interprété comme $\{x \in \mathbb{N} \mid \forall y \in \eta(c), \forall z \in \eta(d), \forall t \in \eta(e), \exists u \in \eta(A), x \cdot y \cdot z \cdot t = u\}$, donc $E_{i,n} = \{x \in \mathbb{N} \mid \exists y \in G_{i,n}, f_{i+1 \bmod n} \cdot f_i \cdot x = e_{i,n} \cdot y\}$ où $G_{i,n} = \{y \in \mathbb{N} \mid e_{i,n} \cdot y = f_i \cdot f_{i+1 \bmod n} \cdot p_i\}$. ($y$ représente un élément de l'interprétation de `eat(I,N)` $\to$ `(tell(fork(I)` $\otimes$ `fork(I+1 mod N))` $\|$ `philosophe(I,N))` du $\texttt{I}^{ème}$ philosophe.)

De plus, la première condition $\forall i, \{p_i\} \subset E_{i,n}$ revient à montrer que $G_{i,n}$ n'est pas vide, ce qui est vrai du fait que $1 \in G_{i,n}$.

La seconde condition (sur $F_{m,p}$) se vérifie par une simple induction sur $x_{m,p}$ et $y_{m,p}$ qui ont été définis exprès. La valuation est donc correcte.

4. Contre-exemple.
Comme $\eta(\texttt{init}) = \{x_{1,n} \cdot y_{1,n}\}$ il nous faut prouver:
$x_{1,n} \cdot y_{1,n} \notin \eta(\texttt{eat(I,N)} \otimes \texttt{eat(I+1 mod N,N)} \otimes \top) = \{x \in \mathbb{N} : \exists a \in \mathbb{N}, x = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a\}$.

Montrons d'abord par induction que: $x_{1,n} \cdot y_{1,n} = f_1 \cdot \ldots \cdot f_n \cdot p_1 \cdot \ldots \cdot p_n$, puis procédons par l'absurde:

Si $x_{1,n} \cdot y_{1,n} = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a$ alors $f_{i+1 \bmod n} \cdot x_{1,n} \cdot y_{1,n} = e_{i,n} \cdot e_{i+1 \bmod n,n} \cdot a \cdot f_{i+1 \bmod n} = f_1 \cdot \ldots \cdot f_{i-1} \cdot f_{i+3} \cdot \ldots \cdot f_n \cdot p_1 \cdot \ldots \cdot p_{i-1} \cdot p_{i+2} \cdot \ldots \cdot p_n \cdot e_{i,n} \cdot e_{i+1 \bmod n,n}$.

En simplifiant on obtient $a \cdot f_{i+1 \bmod n} = f_1 \cdot \ldots \cdot f_{i-1} \cdot f_{i+3} \cdot \ldots \cdot f_n \cdot p_1 \cdot \ldots \cdot p_{i-1} \cdot p_{i+2} \cdot \ldots \cdot p_n$ ce qui est impossible (décomposition en facteurs premiers: $f_{i+1 \bmod N}$ apparaît à gauche du signe $=$ mais pas dans la partie droite, un produit de nombres premiers), cqfd.

Remarque.
On voit bien, sur cet exemple, le rôle de la logique linéaire. En effet si on avait utilisé la logique intuitionniste à la place de ILL et la traduction naturelle des agents CC en séquents, bien qu'on ait les mêmes résultats de correction, le manque au niveau de la complétude n'aurait pas permis d'appliquer cette méthode. En effet, il n'aurait pas été possible de faire la preuve ci-dessus car:

de $\texttt{philosophe(I)} \wedge \texttt{fork(I)} \wedge \texttt{fork(I+1)} \vdash \texttt{eat(I,N)}$

et

$\texttt{philosophe(I+1)} \wedge \texttt{fork(I+1)} \wedge \texttt{fork(I+2)} \vdash \texttt{eat(I+1,N)}$

on peut déduire
$\texttt{philosophe(I)} \wedge \texttt{philosophe(I+1)} \wedge \texttt{fork(I)} \wedge \texttt{fork(I+1)} \wedge \texttt{fork(I+2)} \vdash \texttt{eat(I,N)} \wedge \texttt{eat(I+1,N)}$.

L'espace de phase peut sembler assez complexe et donc peu naturel, cependant on peut le voir comme le monoïde libre engendré par les contraintes atomiques du programme LCC, interprétées comme des singletons, et satisfaisant les contraintes provenant des axiomes non-logiques: des inclusions entre singletons, donc un système d'équations. Cependant un tel espace, basé sur des singletons, ne convient malheureusement pas toujours. Si l'on considère le programme suivant, par exemple: `P = tell(d)`, `Q = c` $\rightarrow$ `P`, une structure de phase basée sur des singletons ne pourra pas prouver que `P` n'atteint jamais un store contenant `c`, i.e. $\texttt{P} \not\vdash \texttt{c} \otimes \top$, car on aura $\eta(\texttt{P}) = \eta(\texttt{c}) \cdot \eta(\texttt{Q})$ pour rester correct vis à vis de la deuxième déclaration.

5.4.2 Exemple 2 - Producteur/Consommateur

Le protocole producteur/consommateur avec m producteurs et k consommateurs qui communiquent via un buffer de taille n peut être codé en LCC comme suit:

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

Rappelons que les producteurs peuvent générer, une par une, des unités de produit quand le buffer n'est pas plein et que les consommateurs peuvent les consommer, une par une, quand le buffer n'est pas vide.

On peut, une fois encore, constater la simplicité du codage réalisé en LCC.

Nous allons prouver deux propriétés importantes sur ce codage d'un protocole classique: d'une part l'absence de deadlock (étreinte fatale), d'autre part la sûreté dans le sens où il ne peut jamais y avoir plus d'unités consommées que d'unités qui ont été produites.

Absence de deadlock

Procédons selon la même méthode:

1.
Il faut d'abord exprimer cette propriété sous la forme: $(X; c; \Gamma) \not\longrightarrow_{LCC} (Y; d; \Delta)$.

On peut voir facilement qu'il ne peut y avoir de deadlock que s'il n'y a plus de P, ou plus de C, ou plus de contraintes consommables (dem et pro).

Il nous faut donc prouver $\texttt{init} \not\longrightarrow_{LCC} \texttt{dem}^{n'} \parallel \texttt{P}^{m'} \parallel \texttt{C}^{k'} \parallel \texttt{pro}^{l'}$, avec soit $n' = l' = 0$ soit $m' = 0$ soit $k' = 0$.

2.
Considérons la structure $P = \mathbb{Z}$, $\mathcal{F} = \mathcal{P}(\mathbb{Z})$, $O = \{\emptyset, \{1\}\}$, c'est bien évidemment une structure de phases, qui est d'ailleurs presque identique à celle de l'exemple précédent.

3.
Définissons alors la valuation suivante:

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

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

Il nous faut vérifier que η est correcte:
$\forall p_1 \in \eta(\texttt{P}), \exists p_2 \in \eta(\texttt{P}), dem \cdot p_1 = pro \cdot p_2$, d'où $\eta(\texttt{P}) \subset \eta$(définition de P).
De même pour C, et $\eta(\texttt{init}) = \eta$(définition de init).

4.
Au lieu d'exhiber un contre-exemple, nous allons à nouveau prouver par l'absurde que l' inclusion $\eta(\texttt{init}) \subset \eta(\texttt{dem}^{n'} \parallel \texttt{P}^{m'} \parallel \texttt{C}^{k'} \parallel \texttt{pro}^{l'})$ est impossible.

Supposons que $\eta(\texttt{init}) \subset \{-5^{n'} \cdot 2^{m'} \cdot 3^{k'} \cdot 5^{l'}, 5^{n'} \cdot 2^{m'} \cdot 3^{k'} \cdot 5^{l'}\}$ En comparant les puissances de 5, 3 et 2 dans ces éléments et dans ceux d'init, on constate que toute autre situation que: $n'+l' = n$, $m' = m$ et $k' = k$ est impossible, et donc qu'il en est de même pour un deadlock ($n'+l' = 0 \neq n$, ou $m' = 0 \neq m$, ou $k' = 0 \neq k$). $\eta(\texttt{init})$ n'est donc inclus dans l'interprétation d'aucun deadlock, et donc init ne peut se réduire sur une telle configuration, cqfd.

Sûreté

Pour vérifier qu'il n'y a jamais plus d'unités consommées que d'unités produites, nous devons modifier légèrement le programme, afin de rendre cette information directement 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
```

Ce type de modification qui consiste à ajouter un *oracle* à un programme afin de pouvoir observer une propriété ou un comportement d'intérêt est très commun dans les techniques de vérification. On voit ainsi souvent l'ajout d'un automate-*oracle* pour faire du *model-checking*.

1.
Il nous faut prouver:

$\texttt{init} \not\longrightarrow_{LCC} \texttt{dem}^{n'} \parallel \texttt{pro}^{l'} \parallel \texttt{P}^m \parallel \texttt{C}^k \parallel \texttt{np} = \texttt{np}_0 \parallel \texttt{nc} = \texttt{nc}_0$
avec $\texttt{nc}_0 > \texttt{np}_0$.

2.
Nous utilisons une fois encore une structure assez simple, $P = \mathbb{Q}$, $\mathcal{F} = \mathcal{P}(\mathbb{Q})$, $O = \{\emptyset, \{1\}\}$.

3.
Et la valuation suivante:

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

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

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

Cette valuation est correcte: $\exists d = 2$ tel que $dem \cdot p = pro \cdot p \cdot d$ et $2^x \cdot d = 2^{x+1}$ d'où $\eta(\texttt{P}) \subset \eta$(définition de P).

4.
Il suffit alors de remarquer que $\texttt{nc}_0 > \texttt{np}_0$ et $\eta(\texttt{init}) \subset \{6^{n'} \cdot 3^{l'} \cdot 5^m \cdot 7^k \cdot 2^{np_0} \cdot 2^{-nc_0}\}$ impliquerait $l' < 0$, ce qui est impossible, cqfd.

Remarque.
Une fois encore IL ne pourrait remplacer ILL car si le système peut se réduire sur un nombre d'unités consommées non nul et peut se réduire sur un nombre d'unités produites nul, alors $\texttt{init} \vdash \texttt{nc} = \texttt{nc}_0 \wedge \texttt{np} = \texttt{np}_0 \wedge A$ avec $\texttt{nc}_0 \neq 0$ et $\texttt{np}_0 = 0$.

5.4.3 Autres exemples - exclusion mutuelle

On a vu dans la section 2.4 qu'il n'y a pas, à priori, d'opérateur de séquentialité ($\bullet$) dans la syntaxe de CC ni de LCC, mais que l'on peut l'y ajouter. Le codage correspondant dépend bien sur de l'agent se trouvant avant le $\bullet$, et on peut vouloir prouver des propriétés indépendantes de cet agent, on peut alors s'appuyer sur la propriété que dans $A_x^\bullet$, $ok(x)$ est ajouté au store si et seulement si $A_x^\bullet$ a terminé son exécution. L'exemple suivant d'exclusion mutuelle par sémaphores illustre l'utilisation de cette méthode pour prouver des propriétés de sûreté pour un programme LCC_{seq}:

$\texttt{P}_i$ = sem $\rightarrow$ A $\bullet$ (sem $\parallel$ $\texttt{P}_i$)
init = sem $\parallel$ $\texttt{P}_1$ $\parallel \cdots \parallel$ $\texttt{P}_N$

On rappelle que A $\bullet$ B est défini comme $\exists x(\texttt{A}_x^\bullet \parallel ok(x) \rightarrow \texttt{B})$.

Les sémaphores sont gérés très simplement en LCC, nous avons donc un parfait exemple de programme LCC_{seq} dont on voudrait prouver la sûreté, c'est à dire que l'exclusion mutuelle fonctionne correctement.

1.
Nous voulons donc prouver que deux sections critiques $ok(x)$ ne peuvent avoir

lieu en même temps: $\forall Z, c, \Gamma, (\emptyset; 1; \texttt{init}) \not\longrightarrow_{LCC} (\{x, y\} \cup Z; c \otimes ok(x) \otimes ok(y); \Gamma)$.

2, 3, 4.
C'est très simple avec la structure $P = \mathbb{N}$ et la valuation suivante:

$$\eta(\texttt{sem}) = \eta(ok(x)) = \eta(\texttt{init}) = \{2\} \text{ and } \eta(\texttt{A}_x^\bullet) = \eta(\texttt{P}_i) = \{1\}$$

La correction de cette valuation et la preuve d'existence d'un contre-exemple (encore une fois par l'absurde) sont laissés comme exercice au lecteur.

5.5 Complétude

Comme souligné dans les exemples précédents, ILL semble être assez lié aux LCC pour être utilisable dans les preuves. Il existe en fait une raison théorique à cela: les résultats de complétude partielle des traductions.

Pour la partie sémantique des phases, la complétude est totale comme le montre le théorème 5.6. Ainsi il y a équivalence entre l'implication linéaire et l'inclusion des faits correspondants. Nous avons déjà utilisé la correction pour dire que s'il y a un contre-exemple alors il n'y a pas implication, mais réciproquement on peut noter que s'il n'y a pas implication, alors il doit exister un contre-exemple (structure de phases, valuation, et contre-exemple pour l'inclusion).

Pour la partie LCC-ILL, rappelons le théorème 3.5 et son corollaire:
La correction est totale, donc s'il n'y a pas implication, il n'y a pas de dérivation.
En revanche on n'a la réciproque que dans le cas d'une contrainte linéaire succès, c'est à dire que $A \vdash c$ implique $A \longrightarrow d$ avec $d \vdash c$, donc si une configuration ne se réduit sur aucune contrainte impliquant c alors sa traduction n'implique pas c.

La mise bout à bout de ces deux résultats nous assure que dans les cas où l'on veut prouver qu'une configuration ne se réduit pas sur un succès c et que cette propriété est vraie, ainsi que le fait que la configuration ne se réduit pas non plus sur une contrainte impliquant c, alors il existe un contre-exemple (structure de phases, valuation, non-implication), donc une preuve

sémantique du type exposé ici.

Ce résultat ne permet cependant pas de traiter toutes les propriétés de sûreté, en particulier le cas des deadlocks, qui sont des suspensions et non pas des succès.

On peut alors utiliser une logique permettant de caractériser des observations encore plus fines: la logique non-commutative. Elle est décrite en annexe C et possède une sémantique des phases. Les résultats de la section 3.4 nous permettent alors d'étendre la propriété précédente au cas des suspensions:

Si une configuration n'a pas de suspension (deadlock) alors sa traduction n'implique aucune traduction de suspension, et il y a donc une preuve sémantique de cette propriété.

Par contre, cette logique qui peut nous conforter dans l'idée de chercher une preuve à telle ou telle propriété, ne nous a, jusqu'à présent rien apporté au niveau des preuves: nous n'avons pas encore d'exemple de propriétés pour lesquelles il existe une preuve non-commutative et pas de preuve commutative. Comme les structures en jeu sont beaucoup plus complexes, nous avons donc préféré ne montrer ici que la version commutative de notre méthode de preuves sémantiques.

5.6 Conclusion

Nous avons donc obtenu, au travers d'une sémantique de la prouvabilité de la logique linéaire, un outil de preuve de propriétés de programmes LCC. Cette méthode semble spécifiquement adaptée aux cas où l'on peut construire facilement un contre exemple et ainsi démontrer une propriété universelle, comme c'est le cas des propriétés de sûreté. Cependant, même lorsque la complétude s'applique, rien ne garantit l'existence d'un contre-exemple *simple*, aussi cette méthode semble-t-elle réservée aux cas où l'on peut facilement s'abstraire du système de contraintes, ce qui est le cas des protocoles de communication décrits dans les exemples de ce chapitre. Afin de généraliser cette méthode à d'autres exemples et même dans le cas d'exemples aussi simples que ceux exposés ici, il serait souhaitable de pouvoir automatiser, au moins partiellement, la recherche de contre-exemples. C'est cette voie que nous décrirons dans le chapitre suivant.

On peut noter que tous les exemples de ce chapitre n'ont pas nécessité les raffinements de la sémantique logique développés au chapitre 4. Cela est bien sur du au fait que nous avons choisi des exemples simples, mais il est

important de réaliser que si nous rajoutons, avec un choix, une branche qui aboutit à un échec, mais que nous voulons prouver des propriétés sur les autres branches, il devient nécessaire de pouvoir différentier l'échec de ses *conséquences*. Une autre situation où les résultats du chapitre 4 sont indispensables se produit si l'on essaye de prouver que dans aucun succès ou store accessible à partir d'un agent A il n'y a *qu'une seule fois* la contrainte c. En effet c fait partie des *conséquences* de $c \otimes c$ au sens de 3.3! Suivant les situations on devra donc choisir une sémantique (ou même, dans le cas des suspensions, une logique) adaptée au type de propriété que l'on souhaite prouver.

Chapitre 6

Phase model-checking

"I don't want to achieve immortality through my work.
I want to achieve it through not dying."
– Woody Allen

6.1 Introduction

Ayant obtenu une méthode originale pour prouver des propriétés de programmes, il est naturel de chercher à l'automatiser, du moins en partie. Nous allons donc voir une possibilité d'automatisation de la recherche d'espaces de phase et de contre-exemples basée sur les outils efficaces que nous apporte la programmation avec contraintes.

Cette approche est basée sur un certain nombre de remarques:

- Nous voulons une méthode de recherche *statique*, nous ne pouvons donc pas utiliser les techniques d'Okada et Terui [27], qui génèrent un espace de phase simple, ce qui est indispensable pour pouvoir raisonner dessus, mais à partir d'une recherche de preuve. On mimerait ainsi en partie la dérivation ce qui n'est bien évidemment pas le but recherché.
- Nous allons donc devoir énumérer des espaces de phases, mais nous ne pouvons bien sur pas tous les essayer, il nous faut donc nous restreindre a une certaine *classe d'espaces* où il sera possible de raisonner facilement (afin de prouver la non-inclusion).
- Dans [38] Yilma a abordé ce type d'approche, mais en essayant de manière systématique toutes les possibilités, dans des structures de taille très réduites (et cycliques).

– Il s'avère que tous les exemples du chapitre 5 ont des solutions dans des structures de phases du même type, où tous les ensembles sont des faits et où la valuation associe des singletons aux formules.

Ces remarques, et en particulier la dernière, qui remplace *inclusion* par *égalité*, nous poussent à essayer de tirer parti des technologies de résolution de contraintes pour obtenir une structure de phases satisfaisante.

6.2 Implémentation

Comme nous l'avons vu avec les exemples du chapitre précédent, $\mathbb{N}$ peut souvent servir comme monoïde de base pour la structure de phase: en effet, nous recherchons en quelque sorte un monoïde libre engendré par la valuation et $\mathbb{N}$ remplira sans problème ce rôle si l'on utilise a bon escient les entiers premiers.

D'autre part nous avons pu constater que dans de nombreux cas une structure finie était suffisante, on peut donc imaginer rechercher des structures de phases dont le monoïde serait fini. On sait d'ailleurs que cette restriction n'en est pas une dans le cas *clos*, puisque Okada et Terui ont montré récemment que IMALL avait la propriété des modèles finis [28].

On se restreindra aussi aux structures de phases où tout les ensembles sont des faits, en effet en plus de nous avoir suffi jusqu'ici, ce type de structures *naïves* selon la terminologie d'Okada, est complet pour un fragment de LL $(\otimes, \oplus, A \multimap \alpha)$ qui n'est certes pas le notre, mais est suffisant pour coder MALL [27].

Pour établir une propriété de sûreté on va alors chercher une valuation correcte, c'est à dire un ensemble de parties de ce monoïde correspondant à chaque contrainte atomique et vérifiant les inclusions correspondants aux axiomes non-logiques.

Si l'on s'intéresse au cas où ces parties sont des singletons, cas dont nous avons certes expliqué qu'il n'était pas complet dans la section 5.4.1, mais qui a permis, comme nous l'avons vu, de résoudre un certain nombre de problèmes, les inclusions mentionnées ci-dessus deviennent des égalités. La recherche d'une valuation consiste alors en une attribution a certaines variables de valeurs dans un domaine fini, de manière à satisfaire certaines contraintes d'égalité.

Il devient naturel de vouloir réaliser cette recherche en utilisant les outils que CC, et même la programmation avec contraintes en général, met à notre disposition: des solveurs sur les domaines finis [8]. Nous avons donc réalisé une

implémentation en GNU-Prolog (voir annexe D) qui extrait les contraintes du programme, les génère et les ajoute au store.

Il nous faut alors trouver notre contre-exemple, c'est à dire satisfaire une nouvelle contrainte, cette fois-ci de non-inclusion donc de non-égalité, correspondant à la propriété de sûreté à démontrer. Si le solveur de contraintes sur les domaines finis y arrive, il peut produire la valuation et le contre exemple nécessaire à la validation du programme.

6.3 Exemples

6.3.1 Dîner des philosophes

Prenons comme exemple une version non-récursive du dîner des philosophes:

```
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).

phil3 :-
    fork3,fork4 -> eat3,(eat3 -> fork3,fork4,phil3).

phil4 :-
    fork4,fork5 -> eat4,(eat4 -> fork4,fork5,phil4).

phil5 :-
    fork5,fork1 -> eat5,(eat5 -> fork5,fork1,phil5).
```

ce programme ayant été écrit pour être utilisé avec le solveur donné en annexe D n'utilise que des caractères ASCII, on a donc utilisé `','` pour $\otimes$ et $\|$, suivant la position, et `'->'` pour $\rightarrow$.

On obtient alors l'exécution suivante:

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

true ?

(1600 ms) yes
| ?-
```

En fait l'exécution se déroule ainsi:

- le fichier est lu et parsé,
- 16 variables correspondant aux contraintes atomiques (cinq pour chaque `phil`, `fork` et `eat` plus une pour `goal`) sont déclarées,
- 10 autres variables correspondant aux *ask* sont déclarées,
- 16 contraintes d'égalité sont ajoutées au solveur: une pour chaque *ask* et une pour chaque déclaration,
- enfin une contrainte de non-égalité est ajoutée pour la propriété.

On aurait donc pour n philosophes, $5n + 1$ variables, $3n + 1$ contraintes d'égalité et 1 contrainte de non-égalité. Des résultats pour des instances plus grandes de ce problème sont résumés dans le tableau 6.3.5.

Remarque.
A priori une simple contrainte `A #\= B` ne permet que de vérifier des propriétés du type $A \not\vdash B$, or on voudrait assez souvent montrer $A \not\vdash B \otimes \top$. C'est ici que l'arithmétique sur les entiers nous aide, en effet si m, n et p

sont des entiers $\forall p, m \neq n \cdot p$ équivaut à m *modulo* $n \neq 0$ or le solveur de contraintes de GNU-Prolog supporte très bien ces contraintes `A rem B #\= 0`.

Si l'on remplace les variables par les contraintes atomiques auquelles elles correspondent, voici le problème de satisfaction de contraintes que le solveur a à résoudre:

```
goal #= phil1 * phil2 * phil3 * phil4 * phil5 *
        fork1 * fork2 * fork3 * fork4 * fork5

phil1 #= fork1 * _11
_11 * fork2 #= eat1 * _12
_12 * eat1 #= fork1 * fork2 * phil1

phil2 #= fork2 * _21
_21 * fork3 #= eat2 * _22
_22 * eat2 #= fork2 * fork3 * phil2

phil3 #= fork3 * _31
_31 * fork4 #= eat3 * _32
_32 * eat3 #= fork3 * fork4 * phil3

phil4 #= fork4 * _41
_41 * fork5 #= eat4 * _42
_42 * eat4 #= fork4 * fork5 * phil4

phil5 #= fork5 * _51
_51 * fork1 #= eat5 * _52
_52 * eat5 #= fork5 * fork1 * phil5

goal rem (eat1 * eat2) #\=0
```

6.3.2 Producteur/Consommateur

Un autre exemple simple est une instance du problème producteur/consommateur de la section 5.4.2:

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

On peut voir ce programme comme cinq producteurs (`p`) et deux consommateurs (`c`), tout le monde communiquant au travers d'un buffer de taille trois, initialement vide (trois cases 'demandes' `dem`).

Il est alors facile de prouver que, par exemple, le buffer ne va jamais déborder: avec

```
find_phase('prod_cons',[i],[([pro,pro,pro,pro],top)])
i=8, c=1, pro=2, dem=2, p=1
```

ou que de nouveaux consommateurs ne peuvent jamais apparaître:

```
find_phase('prod_cons',[i],[([c,c,c],top)])
i=4, c=2, pro=1, dem=1, p=1.
```

De même pour toutes sortes de propriétés de ce type.

Les deux CSP ont en commun les contraintes venant du programme:

```
p #= _1                          _1 * dem #= pro * p

c #= _2                          _2 * pro #= dem * c

init #= dem * dem * dem * p * p * p * p * p * c * c
```

Les deux contraintes correspondant aux propriétés ci-dessus sont:

```
i rem (pro * pro * pro * pro) #\= 0
i rem (c * c * c) #\= 0
```

6.3.3 Algorithme de Peterson

La méthode présentée ici n'est cependant pas complète pour différentes raisons:

La première est que nous avons choisi de nous restreindre à des modèles finis, et que nous sommes donc incapables de traiter convenablement des programmes dont les paramètres sont non bornés. Dans certains cas un nombre infini de valeurs peut être interprété par un seul entier, mais par exemple, pour le dîner des philosophes avec `N` inconnu, il est très difficile de trouver automatiquement un espace de phase comme celui donné dans [11], puisqu'il s'appuie sur une infinité de nombres premiers.

Le choix de ne considérer que des singletons est une autre limitation. En particulier cela engendre des confusions comme pour le programme suivant: `P = tell(d)`, `Q = c` $\rightarrow$ `P`. Une structure de phases basée sur des singletons ne permet pas de prouver que `c` n'est pas accessible de `P`, i.e. `P` $\not\vdash$ `c` $\otimes \top$, puisqu'on peut déduire $[\![\texttt{P}]\!] = [\![\texttt{c}]\!] \cdot [\![\texttt{Q}]\!]$ de la seconde déclaration.

Enfin, il faut noter que l'inversibilité des entiers peut aussi poser des problèmes: en effet puisque $m \cdot p = n \cdot p$ implique $m = n$, on ne peut pas capturer l'idée que la *présence* de p est nécessaire pour aller de m à n. Il y

a donc des difficultés à faire des preuves sur des programmes comme l'algorithme d'exclusion mutuelle de Peterson, qui *vérifie* la valeur d'une variable, sans changer sa valeur, pour permettre à un de ses processus de passer de `off` à `on`.

```
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).

p2 :- off2,turn1 -> on2,turn1,wait2.
p2 :- off2,turn2 -> on2,turn1,wait2.

wait2 :- off1 -> off1,cs2,(cs2,on2 -> off2,p2).
wait2 :- turn2 -> turn2,cs2,(cs2,on2 -> off2,p2).

init :- off1,off2,turn1,p1,p2.
```

Il est cependant important de noter que ces difficultés ne sont pas insurmontables et que l'on peut montrer des propriétés importantes pour de tels programmes, simplement en exprimant plus précisément ce que l'on veut démontrer. Ainsi, si l'on considère à nouveau l'exemple précédent, il est montré dans [38] que s'il n'est pas possible de prouver directement que $\texttt{init} \not\longrightarrow^* \texttt{cs1} \otimes \texttt{cs2} \otimes \top$, on peut remarquer qu'il n'est possible d'atteindre un tel état, qu'en passant par un état d'une des quatre formes suivantes: $\texttt{on1} \otimes \texttt{on2} \otimes \texttt{turn1} \otimes \texttt{turn2} \otimes \top$, $\texttt{on1} \otimes \texttt{on2} \otimes \texttt{turn1} \otimes \texttt{off1} \otimes \top$, $\texttt{on1} \otimes \texttt{on2} \otimes \texttt{off2} \otimes \texttt{turn2} \otimes \top$ et $\texttt{on1} \otimes \texttt{on2} \otimes \texttt{off1} \otimes \texttt{off2} \otimes \top$ puis utiliser la sémantique des phases pour prouver cela.

Nous pouvons donc à notre tour utiliser le *phase model-checker* et obtenir:

```
| ?- 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
cs2=1
wait2=1
on2=2
p2=1
cs1=1
off2=2
```

```
wait1=1
turn2=2
on1=2
off1=2
turn1=2
p1=1

true ?

(1420 ms) yes
```

Ce qui prouve la propriété d'exclusion mutuelle.

6.3.4 Doubletons

On a vu que pratiquement toutes les limitations initiales de notre modèle sont surmontables, cependant il est des cas où se limiter à des singletons est vraiment rédhibitoire, par exemple si l'on considère l'exemple simple du protocole producteur/consommateur donné précédemment et que l'on souhaite prouver qu'un producteur ne peut pas consommer, i.e. $p \parallel pro \not\longrightarrow dem \parallel \Gamma$.

Le *phase model-checker* ne donne aucun résultat:

```
find_phase('prod_cons',[p,pro],[([dem],top)]).
/home/beaune/contraintes/soliman/pl/prod_cons.lcc

(950 ms) no
```

En effet avec des singletons on obtient l'équation $p \cdot pro = p \cdot dem$ à partir de la déclaration de p.

Nous avons déjà mentionné le fait que des solveurs sur les ensembles permettraient de résoudre ce problème, mais il est important de noter qu'en modifiant quelque peu notre implémentation de manière à lui faire attribuer des *doubletons* aux contraintes atomiques à l'aide du constructeur de listes de GNU-Prolog, nous avons déjà à notre disposition un solveur plus général.

Ainsi on a:

```
| ?- find_dbl('prod_cons',[p,pro],[([dem],top)]).
/home/beaune/contraintes/soliman/pl/prod_cons.lcc
p=[1,1]
pro=[2,1]
```

```
dem=[2,2]
c=[0,0]

true ?
(3850 ms) yes
```

On notera qu'il est aisé de faire ainsi opérer notre solveur sur des listes de taille bornée supérieure à deux, mais il devient alors assez lent et le manque d'un solveur spécialisé se fait réellement sentir.

6.3.5 Temps d'exécution

Le tableau suivant résume nos premiers résultats expérimentaux. La taille de l'exemple indique le nombre d'agents initiaux; les nombres de variables et de contraintes indiquent la taille du problème de satisfaction de contraintes généré par le *phase model-checker*.

protocole	taille	nombre de variables	nombre de contraintes	temps pour prouver la sûreté
producteur/consommateur	7	7	5	10 ms
idem avec doubletons	2	12	5	3850 ms
dîner des philosophes	5	26	16	480 ms
dîner des philosophes	10	51	31	3470 ms
dîner des philosophes	20	101	61	44730 ms
algorithme de Peterson	2	25	25	1420 ms

6.4 Conclusion

Ce chapitre ne prétend pas montrer un système achevé de vérification de propriétés de programmes, mais il illustre la possibilité réelle de mettre en oeuvre l'automatisation des techniques du chapitre 5.

L'implémentation assez naïve du vérificateur, dont le code source est disponible dans l'annexe D, a suffi à résoudre un certain nombre de problèmes, mais il faudrait bien sur travailler plus longuement à son élaboration. Par exemple en utilisant d'autres contraintes et d'autres solveurs pour traiter des propriétés différentes, ce serait le cas avec les contraintes universellement quantifiées sur les réels [2].

En effet, il faut souligner que si nous avons fait le choix de nous restreindre, le plus souvent, à des singletons pour les interprétations des contraintes, cela n'est pas suffisant dans le cas général, il serait donc sans doute nécessaire de travailler avec un solveur sur les parties finies de domaines finis pour traiter les cas problématiques.

Une autre des limitations vient de l'aspect non-heuristique de l'énumération, effectuée à la fin de la propagation de contraintes, des stratégies de recherche permettraient sans doute un gain de temps et donc d'efficacité.

Enfin se pose le cas des programmes récursifs avec paramètres que nous n'avons pour l'instant pas réussis à traiter de manière satisfaisante avec cette approche. Des techniques plus fines de calcul formel semblent nécessaire, mais peut-être y perdrait-on le gain apporté par la programmation par contraintes.

Chapitre 7

Conclusion et perspectives

"If you can't convince them, confuse them."
– Harry S. Truman

Notre travail a porté principalement sur deux axes que nous considérons comme très importants de la liaison entre langages CC et logique linéaire: l'élaboration d'une sémantique plus fine, et l'utilisation de la sémantique pour la vérification de programmes.

Plus précisément, nous avons montré, après avoir rappelé l'état de l'art du domaine, comment une traduction plus fine des agents en formules de la logique linéaire était rendue possible par l'enrichissement de la théorie sur laquelle on travaille. Nous avons ainsi aboutit à des résultats aussi larges que ceux du chapitre 3 en ce qui concerne les observables concernées, succès, stores et suspensions, mais plus précis ou plus généraux. La différence est particulièrement remarquable dans le cas de l'observation des suspensions qui se trouve grandement facilitée puisqu'elle n'est plus limitée aux contraintes de synchronisation.

Dans une seconde partie de la thèse, nous avons montré comment la sémantique basée sur la logique linéaire que nous avions décrite dans les chapitres 3 et 4 pouvait déboucher sur des applications directes dans le domaine de la vérification de propriétés de programmes. La méthode en question, basée sur la sémantique des phases de la logique linéaire, a été testée sur des exemples classiques et son automatisation, dans un cadre restreint, a donné lieu à une implémentation décrite au chapitre 6 de la technique de *phase model-checking*.

Ces travaux ont ainsi répondu à certaines questions mais en ont aussi posé un grand nombre.

D'une part du point de vue du langage lui-même, si nous avions déjà de fortes intuitions quant à l'utilisation de LCC et ses apports, en particulier depuis [30, 36], nous avons confirmé les liens entre CC et logique linéaire et donc renforcé l'envie d'utiliser LCC, et si une première implémentation a été réalisée durant cette thèse, il n'en reste pas moins que nous n'en sommes qu'à commencer à comprendre ce que LCC peut nous apporter. Des travaux ont déjà débuté afin de déterminer les gains en terme de complexité (sur les domaines finis) ou d'expressivité (en comparaison avec des paradigmes comme CHR [13] par exemple, en particulier vis à vis de la sémantique opérationnelle décrite dans [1]), mais aucun résultat concluant n'a encore pu être dégagé. Il semble donc qu'il faille en passer par une plus grande utilisation de LCC dans la pratique pour réaliser pleinement comment la combinaison unique de programmation impérative et déclarative qui est à la base de ce paradigme peut être exploitée au mieux.

L'aspect sémantique soulève aussi de nombreuses interrogations; on peut noter en particulier que les preuves des théorèmes de complétude du chapitre 4 ne sont ni très simples, ni très élégantes. Il serait bon de mieux comprendre la liaison même CC-LL afin de peut-être aboutir à des preuves plus simples, ou tout au moins, mieux comprises. On peut aussi se demander si nous avons atteint la limite de ce que l'on pouvait caractériser, ou si d'autres observables d'intérêt nous échappent encore.

Au niveau de la logique, nos recherches nous ont principalement posé deux questions: l'une sur les problèmes liés aux exponentiels, l'autre sur ceux liés aux quantificateurs. En effet nous avons, dans le premier cas, du évacuer complètement les exponentiels de nos sémantiques, ce qui peut certes être perçu comme un gain, mais qui nous interpelle sur les bénéfices exacts qu'apporterait leur réintroduction. Cela nous demanderait une meilleure compréhension des *systèmes de contrainte linéaires* et viendra sans doute avec le temps. Quant aux quantificateurs, il nous a semblé que le premier ordre, s'il est naturel pour la programmation (logique), l'est beaucoup moins pour la logique qui se comporte de manière beaucoup plus naturelle à l'ordre supérieur. Les quantificateurs du premier ordre font intervenir un domaine *extérieur* dans le calcul, et il semble bien difficile de conserver les propriétés intrinsèques de la logique dans ce cadre. Les "affaiblissements" liés aux quantificateurs nous ont amené à des codages artificiels, au niveau sémantique, et l'on pourrait souhaiter avoir une logique qui nous évite ce genre de problèmes.

La sémantique des phases nous a aussi ouvert diverses voies de recherche. Premièrement parce qu'elle a été très peu utilisée, alors qu'elle s'est révélée très pratique pour nous. Il est d'ailleurs à noter que son autre utilisation

principale est l'oeuvre d'Yves Lafont [19, 21, 20] qui utilise lui aussi essentiellement la correction (et pas la complétude) et profite que ce soit seulement une sémantique de la prouvabilité pour s'abstraire de détails inutiles. On pourrait donc envisager, ou du moins espérer, une généralisation de ce type d'utilisation de la sémantique des phases dans d'autres domaines où cette capacité d'abstraction semble nécessaire.

Enfin, nous nous sommes intéressés à une automatisation de la recherche de *certains* espaces de phases bien particuliers: ceux basés sur les entiers naturels, et essentiellement sur des structures de phases à base de singletons. Il n'est pas clair, à l'heure actuelle, que cette restriction soit celle qui convient, ni au niveau des possibilités apportées par les solveurs de contraintes, puisqu'on trouve maintenant des solveurs efficaces sur les réels, y compris avec contraintes universellement quantifiées, ni au niveau de l'expressivité de ce type d'espaces de phases, dont nous n'avons pu prouver la complétude sur aucun fragment *intéressant* de LCC mais qui semble néanmoins utilisable en pratique. Il y a donc encore beaucoup d'expérimentations à faire pour que cette méthode de *phase model-checking* puisse vraiment porter ses fruits, ou au moins montrer ses limites.

Toutes ces voies de recherche ne pourront sans doute pas être explorées par moi, mais j'espère qu'elles susciteront un intérêt, qui sait, peut-être chez vous ...

Annexe A

Logique Linéaire Intuitionniste

Définition A.1 (Formules intuitionnistes) *Les* formules intuitionnistes *sont construites à partir des atomes p, q, ... avec:*

– *les connecteurs multiplicatifs:* $\otimes$ (tenseur) *et* $\multimap$ (implication),

– *les connecteurs additifs:* & (avec) *et* $\oplus$ (plus),

– *le connecteur exponentiel* ! (bien sur ou bang),

– *les constantes:* $\mathbf{1}$, $\top$ *et* $\mathbf{0}$,

– *les quantificateurs: universel* $\forall$ *et existentiel* $\exists$.

Définition A.2 (Séquents intuitionnistes) *Les* séquents *sont de la forme* $\Gamma \vdash A$ *ou* $\Gamma \vdash$, *où* A *est une formule et* Γ *un multi-ensemble de formules.*

Le calcul des séquents est donné par les règles suivantes:

Axiome - Coupure

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

Multiplicatifs

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

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

Additifs

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

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

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

Constantes

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

Bang

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

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

Quantificateurs

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

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

Annexe B

Exemple de programme LCC

Voici un exemple plus conséquent de programme LCC: la reconstruction d'un solveur sur les domaines finis, sans *indexicaux*.

Ce programme a été testé sur la première implémentation que nous ayons faite d'un interpréteur LCC, il a donc une syntaxe un peu lourde avec des listes comme arguments de `lintell` et `linask`, représentant les tenseur de contraintes ou les agents en parallèle.

```
% reconstructing cc(FD) over lcc(H) WITHOUT indexicals

fd(X) :-
    current_prolog_flag(min_integer,M1),
    current_prolog_flag(max_integer,M2),
    lintell([min(X,M1),max(X,M2)]).

get_min(X,M) :-
    linask([Y],[min(X,Y)],[M=Y,lintell([min(X,Y)])]).

get_max(X,M) :-
    linask([Y],[max(X,Y)],[M=Y,lintell([max(X,Y)])]).

set_min(X,M) :-
    linask([Y],[min(X,Y)],[lintell([min(X,M)])]).

set_max(X,M) :-
    linask([Y],[max(X,Y)],[lintell([max(X,M)])]).

% X >= Y + C
```

```
geq_const(X,Y,C) :-
    linask([MinX,MinY],[min(X,MinX),min(Y,MinY),low(MinX,MinY+C)],
   [lintell([min(X,MinY+C),min(Y,MinY)]),geq_const(X,Y,C)]).

% X <= Y + C

leq_const(X,Y,C) :-
    linask([MaxX,MaxY],[max(X,MaxX),max(Y,MaxY),grt(MaxX,MaxY+C)],
   [lintell([max(X,MaxY+C),max(Y,MaxY)]),leq_const(X,Y,C)]).

% X >= Y + Z

geq_var_pos(X,Y,Z) :-
    linask([MinX,MinY,MinZ],[min(X,MinX),min(Y,MinY),min(Z,MinZ),
     low(MinX,MinY+MinZ)],
   [lintell([min(X,MinY+MinZ),min(Y,MinY),min(Z,MinZ)]),
    geq_var_pos(X,Y,Z)]).

% X <= Y + Z

leq_var_pos(X,Y,Z) :-
    linask([MaxX,MaxY,MaxZ],[max(X,MaxX),max(Y,MaxY),max(Z,MaxZ),
     grt(MaxX,MaxY+MaxZ)],
   [lintell([max(X,MaxY+MaxZ),max(Y,MaxY),max(Z,MaxZ)]),
    leq_var_pos(X,Y,Z)]).


% X >= Y - Z

geq_var_neg(X,Y,Z) :-
    linask([MinX,MinY,MaxZ],[min(X,MinX),min(Y,MinY),max(Z,MaxZ),
     low(MinX,MinY-MaxZ)],
   [lintell([min(X,MinY-MaxZ),min(Y,MinY),max(Z,MaxZ)]),
    geq_var_neg(X,Y,Z)]).

% X <= Y - Z

leq_var_neg(X,Y,Z) :-
    linask([MaxX,MaxY,MinZ],[max(X,MaxX),max(Y,MaxY),max(Z,MinZ),
     grt(MaxX,MaxY-M-Z)],
   [lintell([max(X,MaxY-MinZ),max(Y,MaxY),min(Z,MinZ)]),
```

```
    leq_var_neg(X,Y,Z)]).

% Basic User Constraints 'a la' clp(FD)

'x=y+c'(X,Y,C) :-
    geq_const(X,Y,C),leq_const(X,Y,C),
    geq_const(Y,X,-C),leq_const(Y,X,-C).

'x+y=z'(X,Y,Z) :-
    geq_var_neg(X,Z,Y),leq_var_neg(X,Z,Y),
    geq_var_neg(Y,Z,X),leq_var_neg(Y,Z,X),
    geq_var_pos(Z,X,Y),leq_var_pos(Z,X,Y).

'x>=y'(X,Y) :-
    geq_const(X,Y,0),
    leq_const(Y,X,0).

domain(VarList,Lower,Upper) :-
    VarList=[];
    (VarList=[V|L],
     fd(V),
     set_min(V,Lower),
     set_max(V,Upper),
     domain(L,Lower,Upper)).

% SEND MORE MONEY

send_more_money([S,E,N,D,M,O,R,Y]) :-
    domain([S,E,N,D,M,O,R,Y,R1,R2,R3,R4],0,9),
    domain([I1,I2,I3,I4],0,18),
    'x=y+c'(M,R4,0),
    'x+y=z'(D,E,I1),
    'x+y=z'(Y,R1,I1),
    'x+y=z'(N,R,I2),
    'x+y=z'(E,R2,I2),
    'x+y=z'(E,O,I3),
    'x+y=z'(N,R3,I3),
    'x+y=z'(S,M,I4),
    'x+y=z'(O,R4,I4).

% ask (X>Y) -> A  : cc(fd)
```

```
ask_sup(X,Y,G) ->
    linask([MinX,MaxY],[min(X,MinX),max(Y,MaxY),grt(MinX,MaxY)],[G]).
```

Annexe C

Logique Non-commutative

Cette version de la logique linéaire a été développée récemment (voir par exemple [32, 30, 31]). Nous en rappelons ici rapidement le calcul des séquents intuitionniste.

Axiome - Coupure

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

Entropie

$$\frac{\Gamma[\Delta;\Sigma] \vdash A}{\Gamma[\Delta,\Sigma] \vdash A}$$

Connecteurs

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

$$\frac{\Gamma \vdash A \quad \Delta[B] \vdash C}{\Delta[\Gamma; A \multimap B] \vdash C} \qquad \frac{A;\Gamma \vdash B}{\Gamma \vdash A \multimap B}$$

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

Constantes

$$\frac{\Gamma[\Delta] \vdash A}{\Gamma[\Delta;\mathbf{1}] \vdash A} \qquad \frac{\Gamma[\Delta] \vdash A}{\Gamma[\mathbf{1};\Delta] \vdash A} \qquad \vdash \mathbf{1}$$

Quantificateur

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

Bang

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

$$\frac{\Gamma[\Delta] \vdash B}{\Gamma[\Delta; !A] \vdash B} \qquad \frac{\Gamma[\Delta] \vdash B}{\Gamma[!A; \Delta] \vdash B} \qquad \frac{\Gamma[!\Delta, !\Sigma] \vdash B}{\Gamma[!\Delta; !\Sigma] \vdash B}$$

Annexe D

Phase model-checking en GNU-Prolog

```
%% phase.pl : Phase Spaces generation tool for LCC
%% Sylvain Soliman - September 2000

%% find_phase: loads a file and tries to prove property Agents -/-> State * Top
%% find_phase('mutex',[p1,p2,t1],[([t1,t2],top)]).
%% find_phase('prod_cons',[dem,dem,dem,p,p,p,p,p,c,c],
%%            [([pro,pro,pro,pro],top)])
%% find_phase('philo_20',[goal],[([eat1,eat2],top)]).
%% find_phase('peterson',[init],[([cs1,cs2],top)]). NO
%% find_phase('peterson',[init],[([on1,on2,turn1,turn2],top),
%%                               ([on1,on2,turn1,off1],top),
%%                               ([on1,on2,off2,turn2],top),
%%                               ([on1,on2,off1,off2],top)]). YES


find_phase(Infile,Agents,States) :-
    fd_set_vector_max(20),
    myload(Infile,Names,ITime,sgl), % sgl for singleton based interpretation
    add_props(Agents,States,Names,Names2,sgl),
    get_var_list(Names2,Vars,sgl),
    fd_labeling(Vars,[variable_method(smallest)]),
    show_eq_list(Names2),
    cpu_time(ETime),
    Time is ETime - ITime,
    format("time : ~w ms~n",[Time]).
```

```
%% find_dbl: same as before but with doubleton-based interpretation.

find_dbl(Infile,Agents,States) :-
    fd_set_vector_max(20),
    myload(Infile,Names,ITime,dbl),
    add_props(Agents,States,Names,Names2,dbl),
    get_var_list(Names2,Vars,dbl),
    fd_labeling(Vars,[variable_method(smallest)]),
    show_eq_list(Names2),
    cpu_time(ETime),
    Time is ETime - ITime,
    format("time : ~w ms~n",[Time]).

%% myload: reads file Infile and tries to make clauses from it
%% then add constraints corresponding to these clauses

myload(Infile,Names,ITime,Interp) :-
    decompose_file_name(Infile,_,_,Suffix),
    (Suffix = '' ->
     (atom_concat(Infile,'.lcc',File1),
      absolute_file_name(File1,File));
     absolute_file_name(Infile,File)),
    format("~w~n",[File]),
    open(File,read,Sin),
    make_term_list(Sin,L),
    close(Sin),
    cpu_time(ITime),
    enforce(L,[],EqList,[],Names,Interp), %generates Equation List from claus
%    show_eq_list(Names),
%    show_eq_list(EqList),
    fd_solve(EqList).                  % adds constraints to FD store

%% make_term_list: guess what...

make_term_list(Sin,L) :-
    read(Sin,Term),
    (Term = end_of_file ->
     L=[];
     L=[Term|L1],
     make_term_list(Sin,L1)).
```

```
%% enforce: adds equations for each term of the list

enforce([],Eq,Eq,N,N,_).

enforce([H|T],Eq1,EqList,N1,Names,Interp) :-
    enforce_clause(H,Eq1,Eq2,N1,N2,Interp),
    enforce(T,Eq2,EqList,N2,Names,Interp).

%% adds an equation for clause H :- B

enforce_clause(Term,Eq1,Eq2,N1,N2,sgl) :-
    Term =.. [':-'|[H,B]],
    make_goal_list(B,L),
    fd_vector_max(Max),
    fd_domain(Var,0,Max),               % generates new Name for H
    add_binding(N1,H = Var,N3),
    make_eq_list(L,N,Eq1,Eq3,N3,N2),    % generates Eqs qnd names for Body
    member(H = V,N2),
    Eq2 = [V #=# N |Eq3].

%% adds an inclusion for clause H :- B

enforce_clause(Term,Eq1,Eq2,N1,N2,dbl) :-
    Term =.. [':-'|[H,B]],
    make_goal_list(B,L),
    fd_vector_max(Max),
    fd_domain(Var1,0,Max),               % generates new Names for H
    fd_domain(Var2,0,Max),
    add_binding(N1,H = [Var1,Var2],N3),
    make_eq_list_dbl(L,N,Eq1,Eq3,N3,N2), % generates Eqs qnd names for Body
    member(H = [V1,V2],N2),
    Eq2 = [fd_my_element(V1,N),fd_my_element(V2,N)|Eq3].

%% make a list from goal A

make_goal_list(A,L) :-
    var(A) ->
    L = [A];
    (A =.. [Head|Args],
     Head = (',') ->
```

```
     (Args = [H,T],
      L = [H|TL],
      make_goal_list(T,TL));
     L = [A]).

%% returns name N and eq. list Eq2 from list of parallel agents [B] and
%% eq. list Eq1

make_eq_list([B],N,Eq1,Eq2,N1,N2) :-
    make_eq(B,N,Eq1,Eq2,N1,N2).

make_eq_list([B|[H|T]],N,Eq1,Eq2,N1,N2) :-
    make_eq(B,NB,Eq1,Eq3,N1,N3),
    make_eq_list([H|T],NT,Eq3,Eq2,N3,N2),
    N = NB * NT.                    % product of all the names of the list

make_eq(B,N,Eq1,Eq2,N1,N2) :-
    atomic(B) ->
    (Eq2 = Eq1,
     fd_vector_max(Max),
     fd_domain(Var,0,Max),
     add_binding(N1,B = Var,N2),
     member(B = N,N2));                 % constraint or procedure name
    (B =.. [->|[BH,BT]],                % ask
     N = A,
     fd_vector_max(Max),
     fd_domain(A,0,Max),
     make_goal_list(BH,HL),             % list of 'asked' constraints
     add_binding_list(HL,V,N1,N3),      % add bindings and return the product
     make_goal_list(BT,BL),
     make_eq_list(BL,BN,Eq1,Eq3,N3,N2),
%     member(BH = V,N2),
     Eq2 = [A * V #= BN|Eq3]).

%% same for doubletons

make_eq_list_dbl([B],N,Eq1,Eq2,N1,N2) :-
    make_eq_dbl(B,N,Eq1,Eq2,N1,N2).

make_eq_list_dbl([B|[H|T]],N,Eq1,Eq2,N1,N2) :-
    make_eq_dbl(B,NB,Eq1,Eq3,N1,N3),
```

```
    make_eq_list_dbl([H|T],NT,Eq3,Eq2,N3,N2),
    prod_list(NB,NT,N).                  % product of all the names of the list

make_eq_dbl(B,N,Eq1,Eq2,N1,N2) :-
    atomic(B) ->
    (Eq2 = Eq1,
     fd_vector_max(Max),
     fd_domain(Var1,0,Max),
     fd_domain(Var2,0,Max),
     add_binding(N1,B = [Var1,Var2],N2),
     member(B = N,N2));                  % constraint or procedure name
    (B =.. [->|[BH,BT]],                 % ask
     N = [A1,A2],
     fd_vector_max(Max),
     fd_domain(A1,0,Max),
     fd_domain(A2,0,Max),
     make_goal_list(BH,HL),              % list of 'asked' constraints
     add_binding_list_dbl(HL,[V1,V2],N1,N3), % add bindings, return the product
     make_goal_list(BT,BL),
     make_eq_list_dbl(BL,BN,Eq1,Eq3,N3,N2),
%     member(BH = V,N2),
     Eq2 = [fd_my_element(A1 * V1,BN),fd_my_element(A1 * V2,BN),
    fd_my_element(A2 * V1,BN),fd_my_element(A2 * V2,BN)|Eq3]).

prod_list([],_,[]).

prod_list([H|T],L,P) :-
    prod_list(T,L,P1),
    prod_list_one(H,L,P1,P).

prod_list_one(_,[],P,P).

prod_list_one(A,[H|T],P1,P2) :-
    prod_list_one(A,T,P1,P),
    P2 = [A * H|P].

%% pretty print of a list of equations

show_eq_list([]).

show_eq_list([H|T]) :-
```

```
    format("~w~n",[H]),
    show_eq_list(T).

%% add_binding: merge in L the variable binding v=N, returns M

add_binding(L,V = N,M) :-
    member(V = N2,L) ->
    (N = N2, M = L);
    M = [V = N|L].

%% add_binding_list:

add_binding_list([B],V,N1,N2) :-
    fd_vector_max(Max),
    fd_domain(Var,0,Max),
    add_binding(N1,B = Var,N2),
    member(B = V,N2).

add_binding_list([B|[H|T]],V,N1,N2) :-
    add_binding_list([H|T],V1,N1,N3),
    fd_vector_max(Max),
    fd_domain(Var,0,Max),
    add_binding(N3,B = Var,N2),
    member(B = V2,N2),
    V = V2*V1.

add_binding_list_dbl([B],V,N1,N2) :-
    fd_vector_max(Max),
    fd_domain(Var1,0,Max),
    fd_domain(Var2,0,Max),
    add_binding(N1,B = [Var1,Var2],N2),
    member(B = V,N2).

add_binding_list_dbl([B|[H|T]],V,N1,N2) :-
    add_binding_list_dbl([H|T],V1,N1,N3),
    fd_vector_max(Max),
    fd_domain(Var1,0,Max),
    fd_domain(Var2,0,Max),
    add_binding(N3,B = [Var1,Var2],N2),
    member(B = V2,N2),
    prod_list(V1,V2,V).
```

```
%%% get_var_list: extract the list of variables Vars from the list of bindings

get_var_list([],[],_).

get_var_list([_ = V|Names],[V|Vars],sgl) :-
    get_var_list(Names,Vars,sgl).

get_var_list([_ = [V1,V2]|Names],[V1,V2|Vars],dbl) :-
    get_var_list(Names,Vars,dbl).

%% fd_solve: gives the constraints to the FD solver.

fd_solve([]).

fd_solve([H|T]) :-
%    format("--~w~n",[H]),
    call(H),
%    format("++~w~n",[H]),
    fd_solve(T).

%% add_prop: adds constraint Agents -/-> State * Top
%% i.e. Agents mod State =\= 0 if Top=top, Agents =\= State otherwise

add_prop(AgentProd,State,top,Names,sgl) :-
%    make_prod(Agents,P1,Names),
%    format("**   ~w~n",[P1]),
    make_prod(State,P2,Names,sgl),
%    format("**   ~w~n",[P2]),
    AgentProd rem P2 #\= 0.       % #\=# might be necessary if AgentProd is big

add_prop(AgentProd,State,notop,Names,sgl) :-
%    make_prod(Agents,P1,Names),
%    format("**   ~w~n",[P1]),
    make_prod(State,P2,Names,sgl),
%    format("**   ~w~n",[P2]),
    AgentProd #\= P2.

%% same for doubletons

add_prop(AgentProd,State,top,Names,dbl) :-
```

```
    make_prod(State,P2,Names,dbl),
    member(X,AgentProd),
    rem_list(X,P2).

add_prop(AgentProd,State,notop,Names,dbl) :-
    make_prod(State,P2,Names,dbl),
    member(X,AgentProd),
    (member(X,P2) -> fail;true).

rem_list(_,[]).

rem_list(X,[H|T]) :-
    X rem H #\= 0,
    rem_list(X,T).

%% add_props: adds a list of props of the above type.

add_props(Agents,States,N1,N2,Interp) :-
    make_prod(Agents,P1,N1,Interp),
    add_props_prod(P1,States,N1,N3,Interp),
    put_ahead(Agents,N3,N2).

add_props_prod(_,[],N,N,_).

add_props_prod(P1,[(S,Top)|T],N1,N2,Interp) :-
    add_prop(P1,S,Top,N1,Interp),
    put_ahead(S,N1,N3),
    add_props_prod(P1,T,N3,N2,Interp).

%% make_prod: P is the product '*' of the Names corresponding to the elements
%% of list given as first argument.

make_prod([A],P,Names,_) :-
    member(A = P,Names).

make_prod([A|[H|T]],P,Names,sgl) :-
    member(A = V,Names),
    make_prod([H|T],P2,Names,sgl),
    P = V * P2.

make_prod([A|[H|T]],P,Names,dbl) :-
```

```
    member(A = V,Names),
    make_prod([H|T],P2,Names,dbl),
    prod_list(V,P2,P).

%% put_ahead: reorder the second argument so as to have the elements of the
%% first list first.

put_ahead([],L,L).

put_ahead([H|T],L1,L2) :-
    select(H = V,L1,L3),
    delete(T,H,T2),
    put_ahead(T2,L3,L4),
    L2 = [H = V|L4].

%% fd_my_element: forces the first argument to equal one of the elements of the
%% list that constitutes the second argument

fd_my_element(_,[]) :- fail.

fd_my_element(X,[H|T]) :-
    X #= H;
    fd_my_element(X,T).
```

Bibliographie

[1] S. Abdenhader. Operational semantics and confluence of constraint propagation rules. In *Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming, Linz*, volume 1330 of *LNCS*, pages 252–266. Springer-Verlag, 1997.

[2] F. Benhamou and F. Goualard. Universally quantified interval constraints. In *Proceedings of CP'2000, 6th International Conference on Principles and Practice of Constraint Programming, Singapore*, volume 1894 of *LNCS*, pages 67–82. Springer-Verlag, January 2000.

[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 removal. In *Proceedings of Coordination*, LNCS. Springer-Verlag, 1997.

[5] G. Boudol. Asynchrony and the π-calculus. Technical report, INRIA, 1992.

[6] M. Carlsson. Freeze, indexing, and other implementation issues in the wam. In Jean-Louis Lassez, editor, *Proceedings of ICLP'87, International Conference on Logic Programming, Melbourne*, MIT Press Series in Logic Programming, pages 40–58, 1987.

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

[8] P. Codognet and D. Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3), June 1996.

[9] A. Colmerauer. *PROLOG II: Manuel de réference et modèle théorique.* Groupe d'Intelligence Artificielle, Faculté, des Sciences de Luminy, 1982.

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

[11] 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.

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

[13] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*, pages 90–107. Springer-Verlag, 1995.

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

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

[16] 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.

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

[18] R. Jagadeesan, V. Shanbhogue, and V.A. Saraswat. Angelic nondeterminism in concurrent constraint programming. Technical report, Xerox Parc, 1991.

[19] Y. Lafont. The undecidability of second order linear logic without exponentials. *Journal of Symbolic Logic*, 61(2):541–548, 1996.

[20] Y. Lafont. The finite model property for various fragments of linear logic. *Journal of Symbolic Logic*, 62(4):1202–1208, 1997.

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

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

[23] K. Marriott M. Falaschi, M. Gabbrielli and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.

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

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

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

[27] 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.

[28] M. Okada and K. Terui. The finite model property for various fragments of intuitionistic linear logic. *Journal of Symbolic Logic*, 2001. To appear.

[29] 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.

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

[31] P. Ruet. Non-commutative logic II : sequent calculus and phase semantics. *Mathematical Structures in Computer Science*, 10(2), 2000.

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

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

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

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

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

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

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