Logique linéaire et algèbres de processus

David Baelde
sous la direction de
Dale Miller
LIX

Mars à Juillet 2005

1 Introduction

L'informatique a besoin de langages élémentaires expressifs et bien compris, afin de raisonner formellement sur un programme, que ce soit pour le compiler ou le certifier.

S'inspirant de la logique, l'informatique théorique a su construire et comprendre des calculs simples comme le λ-calcul, fondation de la programmation fonctionnelle. On se tourne désormais vers de nouveaux défis, notamment le calcul distribué, mais aussi la programmation impérative. Le π-calcul et ses variations permettent d'exprimer ces deux paradigmes, tout en étant très élémentaire. La théorie de la démonstration inspire la recherche sur les algèbres de processus, avec la logique linéaire. De nombreux travaux lient les deux formalismes, de façon plus ou moins significative.

Nous étudions ici les connections entre logique linéaire et différents calculs de processus, dans la tradition de la recherche de preuve. Nous consacrons la Section 2 à un parcours plus détaillé des travaux effectués, dans plusieurs grands courants. Nous travaillons sur CCS asynchrone fini, puis exposons un encodage du Join-calcul, démontrant l'expressivité de l'approche. Nous ne nous restreignons pas aux calculs asynchrones, mais introduisons aussi de nouveaux outils logiques afin d'étendre l'expressivité calculatoire au synchronisme, en encodant CCS fini. Dans la Section 4, nous explorons rapidement quelques applications et enseignements de nos travaux.

Remerciements

Avant tout, je remercie tout le monde au LIX pour m'avoir accueilli, en particulier Dale, les équipes Comète et Parsifal, Romain et Sylvain. Et je me réjouis de travailler encore avec eux ces prochaines années. Un grand merci à Daniel, Gim, Jade, Sam et d'autres pour des tas de discussions enrichissantes ou pas. Merci à tous ceux avec qui j'aime passer mon temps sans parler de logique ni calcul, en particulier Estelle et les pyrénéens que j'ai retrouvés avec joie à Paris cette année. Et merci à tous ceux qui se sentent oubliés dans cette section. Enfin, je suis reconnaissant à toute une communauté d'informaticiens au sens large, pour de nombreux outils de qualité qu'ils ont bien voulu développer comme logiciels libres.

2 Logique et calcul : un aperçu

Il existe plusieurs connexions entre logique et calcul, et il n'est pas toujours aisé de les distinguer.

Une approche très répandue est d'interpréter les preuves comme des programmes. Par exemple, le paradigme *proof-as-program* est incarné par le fameux isomorphisme de Curry-Howard. L'élimination des coupures dans une logique est interprétée comme la réduction du calcul. La correspondance est fructueuse : le programme prouve son type ; et le type spécifie le programme, plus ou moins fortement. Suivant cette approche, Emmanuel Beffara [Bef05] a récemment proposé un système de type linéaire pour une algèbre de processus inspirée des fusions.

La seconde grande école est d'identifier recherche de preuve et calcul. C'est la programmation logique, le paradigme *proof-search-as-computation*. Typiquement, on se restreint à un ensemble complet de preuves, et son énumération constitue le calcul. Par exemple, on ne cherche que des preuves sans coupure, ou focalisées [And92]. La structure du calcul dépend du fragment de logique considérée et peut-être complexe, comme pour le fameux ancêtre Prolog.

Suivant cette approche, Miller [Mil93] a montré que la dynamique du π-calcul peut être encodée dans la logique linéaire. Les processus sont encodés dans des termes, et la réduction est axiomatisée dans la logique. Le présent travail est d'étudier des connexions plus élémentaires, en établissant des encodages purement logiques. On notera que [Mil93] définit bien un langage de programmation logique en généralisant la notion de preuve uniforme [MNPS91] aux séquents à conclusions multiples. Nous ne traiterons pas ce type de problème ici, bien que la question de la frontière entre logique et calcul soit très intéressante.

Nous identifions donc formules et processus, calcul et recherche de preuve, comme Kobayashi et Yonezawa pour ACL [KY93], ou Okada et Terui pour CCS fini asynchrone [OT96]. On se restreint à une stratégie de recherche de preuve dans laquelle le *backtracking* est négligeable, et où on admet l'indéterminisme. De plus, nous considérons des dérivations ouvertes plus que des preuves. Ainsi, nous placons la comparaison entre logique et calcul directement au niveau de la réduction. Au contraire, [KY93] établit la correspondance entre une preuve et la terminaison d'un programme, son calcul étant équipé de la constante $\top$, équivalent du canal w en sémantique par tests. Le calcul d'Okada et Terui est plus proche de CCS et ne dispose donc pas d'une action de terminaison, peu réaliste dans un environnement concurrent. Ils établissent la correspondance entre la possibilité pour P d'effectuer une certaine trace $a_1 \dots a_n$ et la prouvabilité du séquent $[P] \vdash a_1 \otimes \cdots \otimes a_n \otimes \top$, n'exprimant toujours pas la connection plus profonde entre les deux dynamiques.

3 Spécification de calculs en logique linéaire

3.1 Logique linéaire

La logique linéaire [Gir87] peut être présentée succintement comme une logique traitant de ressources. Contrairement aux systèmes usuels, la contraction et l'affaiblissement des hypothèses n'est pas autorisé. Typiquement, si A implique B et A implique C, alors A n'implique pas forcément B et C. Une instance de A est consommée par l'implication. Du refus de la contraction et de l'affaiblissement résulte une duplication des connecteurs de conjonction et disjonction. Pour les *multiplicatifs* $\otimes$ et $⅋$, on sépare l'environnement en deux parties, les ressources consommées par une opérande ne pouvant pas servir à l'autre. Les opérandes des additifs & et $\oplus$ disposent au contraire chacune d'une copie de l'environnement. D'autre part, la négation est remplacée par la notion de dualité. La dualisation est une involution définie sur les formules, la "négation" de p est son dual $p^\perp$. Ces grands traits font de la logique linéaire un système élégant hautement symétrique, bien plus qu'un simple système traitant de ressources. Cette logique a aussi de nombreuses applications en informatique, du λ-calcul aux algèbres de processus.

Les règles d'inférence de la logique linéaire multiplicative sont données Figure 1. Dans un premier temps, seul ce fragment sera utilisé. La logique multiplicative additive est obtenue par l'ajout des connecteurs additifs, dont la définition est donnée Figure 4. On trouvera enfin Figure 2 les règles d'inférence pour les exponentielles, deux modalités permettant d'exprimer la réplication, et de retrouver ainsi la logique usuelle, notamment l'implication :

$$A \Rightarrow B \equiv !A \multimap B \equiv (!A)^\perp \parr B \equiv ?A^\perp \parr B$$

Dans la suite, nous travaillons sous l'hypothèse de distinction des positifs et négatifs : $\forall a, a \neq a^\perp$.

$$\boxed{\begin{array}{l} A^{\perp\perp} = A \qquad \perp^\perp = \mathbf{1} \\ (A \parr B)^\perp = A^\perp \otimes B^\perp \end{array}}$$

$$\frac{}{\vdash A, A^\perp}\ init$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, \perp}\ \perp \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}$$

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

FIG. 1 – La logique linéaire

Définition 1 (Dérivation, dérivation ouverte et n-ouverte, feuille impropre)**.** On suppose donnée une logique, sous la forme d'un ensemble de règles d'inférence. Une *dérivation* ou *preuve* est un arbre dont les noeuds sont étiquetés par des séquents, tel que l'ensemble des fils d'un séquent Γ sont les prémisses d'une instance d'une règle d'inférence dont la conclusion est Γ. On appelle règle *initiale* une règle n'ayant pas de prémisse. On écrit $\vdash \Gamma$ quand il existe une dérivation de racine Γ.

Une *dérivation ouverte* est une dérivation dont certains noeuds n'ont pas de fils, sans pour autant être des instances de règles initiales. De tels noeuds sont appelés *feuilles impropres*. On dit qu'une dérivation est n-ouverte si elle a n feuilles impropres exactement.

On s'intéresse particulièrement aux dérivations 1-ouvertes. Une telle dérivation, de conclusion Δ et de feuille impropre Γ, sera notée :

$$\frac{\vdash \Gamma}{\vdash \Delta}$$

Comme toute logique qui se respecte, la logique linéaire permet d'éliminer les coupures, c'est à dire que la règle suivante est admissible :

$$\frac{\vdash A, \Gamma \quad \vdash A^\perp, \Delta}{\vdash \Gamma, \Delta}\ cut$$

Il faut cependant remarquer que ceci n'est pas vrai dans le cas de dérivations ouvertes, dans lesquelles le processus de normalisation peut rester bloqué en attente sur une feuille impropre. Malgré cela, on ne considère pas la coupure dans notre travail, et les dérivations ouvertes seront toujours sans coupures. La recherche de preuve est toujours une recherche de preuve sans coupure.

3.2 CCS fini asynchrone

Afin de donner une idée simple de notre approche, nous commencons par encoder CCS fini asynchone [Mil80, HT91]. Nous traiterons ensuite du Join-calcul, plus complet. Mais les techniques utilisées ainsi que les démonstrations resteront essentiellement les mêmes.

La syntaxe du calcul est la suivante :

$$P ::= \quad 0 \quad | \quad \bar{a} \quad | \quad a.P \quad | \quad (P \,|\, P)$$

0 est le processus nul, terminé. $a.P$ représente un processus en attente d'une réception sur a, qui deviendra P après réception. $\bar{a}$ est une émission sur a. Puisqu'on se place dans le cas asynchrone, l'émission n'a pas de continuation. Enfin, la composition parallèle permet de faire interagir plusieurs processus, s'exécutant concurremment.

Les termes sont identifiés modulo associativité et commutativité de la composition parallèle, et neutralité de 0. La seule règle de réduction est :

$$a.P \,|\, \bar{a} \,|\, C \ \longrightarrow\ P \,|\, C$$

Définition 2 (Transition étiquetée, action, trace)**.** On présente parfois le calcul en définissant les transformations d'un processus par des *transitions étiquetées* $P \to^a Q$. Ce point de vue considère P en interaction avec l'extérieur, par exemple si l'*action* a est une réception, la transition signifie que P effectue une réception sur a, l'emetteur du message étant extérieur. Une action spéciale notée τ dénote une réduction interne. On dit que P peut effectuer la *trace* $a_1 \ldots a_n$ s'il existe une transition étiquetée $P \to^{a_1} P_1 \to^{a_2} \cdots \to^{a_n} P_n$. Nous n'utiliserons pas cette présentation mais y ferons seulement parfois référence.

Nous encodons les processus dans MLL comme suit :

$$\begin{array}{rcl} [0] & = & \perp \\ [a.P] & = & a^\perp \otimes [P] \\ [\overline{a}] & = & a \\ [P \,|\, Q] & = & [P] \parr [Q] \end{array}$$

Pour traiter les problèmes d'équivalence logique, on utilise la notion de chauffage, proposée pour la CHemical Abstract Machine [BB92].

Définition 3 (Chauffage)**.** Soit Γ un multi-ensemble de formules. On note $\Gamma^\uparrow$ le multiset résultant de l'application maximale des règles synchrones : $\parr$ et $\perp$. $\Gamma^\uparrow$ est appelé le chauffage de Γ.

Théorème 1 (Réduction & Déduction)**.** *Il y a équivalence entre déduction dans MLL pour les encodages de processus, et réduction dans le CCS asynchrone fini. D'une part, à toute réduction correspond une dérivation ouverte. D'autre part pour toute dérivation* 1*-ouverte de conclusion* $[P]$*, la feuille impropre est l'encodage d'un processus Q modulo chauffage, et P se réduit bien vers ce processus.*

$$\forall P\,.\,\forall Q\,.\quad P \to^* Q \quad\Rightarrow\quad \dfrac{\vdash [Q]^{\uparrow}}{\vdash [P]} \tag{1}$$

$$\forall P\,.\quad \dfrac{\vdash \Gamma}{\vdash [P]} \quad\Rightarrow\quad \begin{cases} \exists Q\,.\,\Gamma^{\uparrow} = [Q]^{\uparrow} \\ \forall Q\,.\,\Gamma^{\uparrow} = [Q]^{\uparrow} \Rightarrow P \to^* Q \end{cases} \tag{2}$$

Ce type de résultat est la clé du présent travail. Nous obtenons un théorème équivalent pour le Join-calcul, puis pour le CCS fini synchrone. Le seul paradigme non abordé est le choix. Le connecteur $\oplus$ encode bien une sorte de choix, mais qui ne correspond pas au choix usuel en CCS où le processus choisi doit pouvoir évoluer. Par exemple, on n'a pas $\perp \oplus P \equiv P$.

Deux résultats intermédiaires simples vont nous mener à la démonstration du Théorème 1.

Lemme 1. *Pour toute famille de processus* $(P_i)_{i\leq n}$ *et atomes a et b :*

$$\nvdash a^{\perp}, b^{\perp}, [P_i]_{i\leq n}$$

Démonstration. On démontre par induction que toute dérivation ouverte de conclusion $[P_i]_{i\leq n}, a^{\perp}, b^{\perp}$ a au moins une feuille impropre contenant deux atomes dualisés. Ceci fait, on conclut aisément, car la règle *init* ne peut être utilisée pour fermer une dérivation ouverte, puisqu'elle ne peut agir sur une feuille contenant deux atomes dualisés.

La dérivation ouverte vide est composée de la feuille $[P_i]_{i\leq n}, a^{\perp}, b^{\perp}$ et contient donc au moins deux atomes dualisés. Les règles $\perp$ et $⅋$ ne changent pas cette propriété. La règle *init* ne peut faire disparaitre les feuilles qui nous préoccupent. Enfin, si $\otimes$ est appliquée sur $c^{\perp} \otimes [A]$, soit le nouvel atome dualisé $c^{\perp}$ se retrouve avec un des deux précédents dans la sous-dérivation gauche, soit les deux précédents sont toujours ensemble dans la dérivation droite. Dans les deux cas il reste une feuille surchargée en atomes dualisés. □

Lemme 2. *Pour toute famille de processus* $(P_i)_{i\leq n}$ *:*

$$\nvdash [P_i]_{i\leq n}$$

Démonstration. Supposons par l'absurde qu'il existe de tels séquents prouvables, et considérons une dérivation minimale parmis ces preuves. Elle ne peut être réduite à la règle initiale, car $a^{\perp}, a$ n'est pas un encodage de processus. Elle ne peut commencer par $⅋$ ni $\perp$ par minimalité. Enfin elle ne peut commencer non plus par la règle $\otimes$, puisque la sous-dérivation droite formerait alors une dérivation plus petite de la même forme, contredisant encore la minimalité. On a la contradiction : aucune règle ne peut commencer cette preuve. □

Démonstration du théorème 1.

(1) Cette direction ne pose pas de problème, on procède par induction sur la longueur de la réduction. Le cas de base est trivial. Supposons ensuite que l'on a :

$$P_0 \to^* a.P \,|\, \bar{a} \,|\, C \to P \,|\, C \quad \text{et} \quad \dfrac{\vdash a^{\perp} \otimes [P], a, [C]^{\uparrow}}{\vdash [P_0]}$$

On construit alors la dérivation suivante, puis on la complète par chauffage et on obtient bien une dérivation de feuille impropre $[P\,|\,C]^{\uparrow}$:

$$\dfrac{\dfrac{\dfrac{}{\vdash a^{\perp}, a} \quad \vdash [P], [C]^{\uparrow}}{\vdash a^{\perp} \otimes [P], a, [C]^{\uparrow}}}{\vdash [P_0]}$$

(2) Remarquons d'abord que toute dérivation 1-ouverte de conclusion un encodage de processus a encore un encodage comme feuille impropre. En effet, les dérivations gauches issues des $\otimes$ ne peuvent être ouvertes, d'apres le Lemme 2 on a déja forcément une feuille impropre à droite.
Montrons maintenant que l'on peut se ramener à une dérivation où les sous-dérivations gauches sont réduites à une application de la règle *init*. Pour cela on procède par induction, en faisant décroitre la taille maximum d'une sous-dérivation gauche, c'est à dire le nombre de règles utilisées dans cette dérivation. Quand la mesure vaut 1, la dérivation ouverte représente trivialement une réduction dans CCS, les applications de $\otimes$ représentant bien des synchronisations puisque *init* apparait immédiatemment à gauche, et les applications de $\parr$ et $\perp$ ne sont que des manipulations syntaxiques vides.
Reste à montrer comment on fait décroitre cette mesure. Supposons que l'on a une application de $\otimes$ avec une dérivation gauche non triviale :

$$\dfrac{\dfrac{\Theta}{\vdash a^{\perp}, \Gamma} \quad \vdash [P], \Delta}{\vdash a^{\perp} \otimes [P], \Gamma, \Delta}$$

Énumérons les premières règles possibles dans Θ :
$\parr$: Γ contient une composition parallèle, qui peut être explosée avant le $\otimes$.
$\perp$: Γ contient un $\perp$, qui peut être supprimé avant le $\otimes$.
$\otimes$: Γ contient une formule de la forme $b^{\perp} \otimes [Q]$. Après la nouvelle application de $\otimes$, $a^{\perp}$ ne peut aller du même côté que $b^{\perp}$, d'après le Lemme 1. Il est donc placé avec $[Q]$ et l'on peut ainsi tirer la deuxième application de $\otimes$ avant la première :

$$\dfrac{\dfrac{\dfrac{\Theta_1}{\vdash b^{\perp}, \Gamma_1} \quad \dfrac{\Theta_2}{\vdash a^{\perp}, [Q], \Gamma_2}}{\vdash a^{\perp}, b^{\perp} \otimes [Q], \Gamma_1, \Gamma_2} \quad P, \Delta}{\vdash a^{\perp} \otimes [P], b^{\perp} \otimes [Q], \Gamma_1, \Gamma_2, \Delta} \quad \longrightarrow \quad \dfrac{\dfrac{\dfrac{\Theta_1}{\vdash b^{\perp}, \Gamma_1}}{\vdash b^{\perp} \otimes [Q], \Gamma_1} \quad \dfrac{\dfrac{\Theta_2}{\vdash a^{\perp}, Q, \Gamma_2} \quad P, \Delta}{\vdash a^{\perp} \otimes [P], Q, \Gamma_2, \Delta}}{\vdash a^{\perp} \otimes [P], b^{\perp} \otimes [Q], \Gamma_1, \Gamma_2, \Delta}$$

□

On peut se demander pourquoi l'on s'intéresse aux dérivations ouvertes et pas à des preuves. En fait, le théorème 1 n'est tout simplement plus valide si on considère comme équivalent du calcul $P \to^{*} Q$ une preuve terminée de $[Q] \vdash [P]$. En effet, cette présentation permet une interaction entre P et Q, autrement dit entre le passé et le présent. Par exemple, on aurait la réduction $\overline{b}\,|\,a.b.0 \to^{*} a.0$:

$$\dfrac{\dfrac{\dfrac{}{\vdash a, a^{\perp}} \qquad \dfrac{\dfrac{}{\vdash b^{\perp}, b} \quad \dfrac{\dfrac{}{\vdash \mathbf{1}}}{\vdash \mathbf{1}, \perp}}{\vdash \mathbf{1}, a^{\perp} \otimes b^{\perp} \otimes \perp, b}}{\vdash a, \mathbf{1}, a^{\perp} \otimes b^{\perp} \otimes \perp, b}}{a^{\perp} \otimes \perp \vdash a^{\perp} \otimes b^{\perp} \otimes \perp, b}$$

3.3 Join-calcul

$$(!A)^{\perp} = ?(A^{\perp})$$

$$\dfrac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ ?C \qquad \dfrac{\vdash \Gamma}{\vdash \Gamma, ?A}\ ?W$$

$$\dfrac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ ?D \qquad \dfrac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\ !$$

FIG. 2 – Les exponentielles

Le Join-calcul [FG96] est un calcul de processus asynchrone moins élémentaire que CCS ou le π-calcul. Sa conception plus structurée privilégie en effet l'implémentation. Le calcul n'en reste pas moins aussi expressif que le π-calcul, donc Turing-puissant. Il dispose de passage de nom, de réplication et de création dynamique de noms frais. L'encoder dans les formules de la logique linéaire est donc un défi intéressant. D'ailleurs, la présentation même de la sémantique opérationnelle du calcul appelle fortement a l'encodage dans la logique linéaire.

Définissons la syntaxe du Join-calcul. Il faut distinguer trois classes : les processus P, les définitions D et les motifs Join J :

$$\begin{array}{rcl} P & ::= & 0 \quad | \quad P\,|\,P \quad | \quad a(\overline{x}) \quad | \quad \textbf{let def}\, D \,\textbf{in}\, P \\ D & ::= & J \triangleright P \quad | \quad D \wedge D \\ J & ::= & a(\overline{x}) \quad | \quad J\,|\,J \end{array}$$

Intuitivement, les définitions sont des règles de réécriture, mais la prémisse n'est composée que d'atomes $a(\overline{x})$ alors que la conclusion est un processus à part entière, qui peut notamment engendrer de nouvelles règles. Par rapport au π-calcul, la définition joue à la fois le rôle de la réplication et de la création de nom frais — ou extrusion de contexte. En effet, une définition n'est pas détruite par son utilisation, et les noms de canaux utilisés en réception, noté $dv(D)$, est rendu frais lors de l'introduction de la définition.

On définit aussi l'ensemble des variables de reception, noté $rv(J)$, utilisé pour l'instantiation consécutive à l'application d'une règle. Il y a deux contraintes. D'abord l'instantiation doit être complète, ce qui exclut les motifs exprimant la réception d'une valeur prédéfinie, comme **let def** $a(3) \triangleright P$ **in** Q. Nous verrons que cette contrainte n'est en un sens pas aussi artificielle qu'il y parait, puisqu'elle trouve en partie écho dans notre cadre de travail. D'autre part les motifs sont contraints à ne recevoir qu'une fois sur chaque variable de réception, ce qui exclut par exemple **let def** $a(x,x) \triangleright P$ **in** Q or **let def** $a(x)|b(x) \triangleright P$ **in** Q.

$$\begin{array}{rclrcl} dv(a(\overline{x})) & = & a & rv(a(\overline{x_i})) & = & \{x_i\} \\ dv(J_1 \,|\, J_2) & = & dv(J_1) \cup dv(J_2) & rv(J_1 \,|\, J_2) & = & rv(J_1) \uplus rv(J_2) \\ dv(D_1 \wedge D_2) & = & dv(D_1) \cup dv(D_2) & & & \end{array}$$

Les règles de réduction du calcul sont données dans le formalisme de la *CHemical Abstract Machine* (CHAM) [BB92]. De part et d'autre du $\vdash$ on omet le contexte inchangé. Comme précisé plus haut, σ_{dv} (resp. σ_{rv}) doit être une substitution dont le domaine est *exactement* $dv(D)$ (resp. $rv(J)$).

$$\begin{array}{rcll} \vdash P \,|\, Q & \rightarrow & \vdash P, Q & \\ \vdash 0 & \rightarrow & \vdash & \\ \vdash \textbf{let def}\, D \,\textbf{in}\, P & \rightarrow & D\sigma_{dv} \vdash P\sigma_{dv} & \text{with fresh } \mathit{range}(\sigma_{dv}) \\ D_1 \wedge D_2 \vdash & \rightarrow & D_1, D_2 \vdash & \\ J \triangleright P \vdash J\sigma_{rv} & \rightarrow & \vdash J \triangleright P \vdash P\sigma_{rv} & \end{array}$$

Voici un exemple de réductions successives :

$$\begin{array}{rcl} & \vdash & \textbf{let def}\, (♀ \,|\, ♂) \triangleright (♀ \,|\, ♂ \,|\, ☺) \,\textbf{in}\, ♀ \,|\, ♂ \\ (♀ \,|\, ♂) \triangleright (♀ \,|\, ♂ \,|\, ☺) & \vdash & ♀ \,|\, ♂ \\ (♀ \,|\, ♂) \triangleright (♀ \,|\, ♂ \,|\, ☺) & \vdash & ♀ \,|\, ♂ \,|\, ☺ \\ (♀ \,|\, ♂) \triangleright (♀ \,|\, ♂ \,|\, ☺) & \vdash & ♀ \,|\, ♂, ☺ \\ (♀ \,|\, ♂) \triangleright (♀ \,|\, ♂ \,|\, ☺) & \vdash & ♀ \,|\, ♂ \,|\, ☺, ☺ \end{array}$$

L'encodage dans les formules de logique linéaire n'est pas surprenant. On remarquera que l'encodage de la définition est symétrique. D'autre part, nous utilisons désormais les quantificateurs : universel pour la fraîcheur des noms ; existentiel pour la réception avec passage de valeur. Nous discuterons plus tard des questions relatives au typage levées par cette représentation.

$$\begin{array}{rcl} [0] & = & \bot \\ [P \,|\, Q] & = & [P] \parr [Q] \\ [a(\overline{x})] & = & a(\overline{x}) \\ [\textbf{let def}\, D \,\textbf{in}\, P] & = & \forall \overline{dv(D)}\,.\,[D] \parr [P] \\ [D_1 \wedge D_2] & = & [D_1] \parr [D_2] \\ [J \triangleright P] & = & ?(\exists \overline{rv(J)}\,.\,[J]^\perp \otimes [P]) \end{array}$$

Voici comment se déroule le calcul de **let def** $a(x) \triangleright a_1(x)|a_2(x)$ **in** $a(x)$ dans la logique. On omet la signature, enrichie du nom frais a après application de la règle $\forall I$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{a(c)^{\perp}, a(c)} \qquad \dfrac{\vdots}{\vdash ?(\exists x \,.\, \dots), a_1(x) \parr a_2(c)}}{\vdash ?(\exists x \,.\, \dots), a(c)^{\perp} \otimes (a_1(c) \parr a_2(c)), a(c)}\ \otimes}{\vdash ?(\exists x \,.\, \dots), \exists x \,.\, (a(x)^{\perp} \otimes (a_1(x) \parr a_2(x))), a(c)}\ \exists I}{\vdash \forall a \,.\, ?(\exists x \,.\, (a(x)^{\perp} \otimes (a_1(x) \parr a_2(x))) \parr a(c))}\ \forall I, ?C, ?D, \parr
$$

Avant d'aller plus loin, il faut préciser ce que signifie l'encodage des réceptions, $\exists x \,.\, c(x)^{\perp} \otimes P$. Habituellement, le π-calcul n'est pas typé. Tout est nom et canal à la fois. Il est alors impossible de typer dans la logique une formule comme $a(x)$. Ceci sera en fait vu comme une notation pour $carry(a, x)$ ou $carry$ est un symbole de prédicat distingué, de type $nom \to nom \to o$[1]. Malheureusement, on n'a alors plus un encodage purement logique.

Si l'on veut représenter un π-calcul typé, on se passe du prédicat $carry$ en représentant simplement un canal x de type t par un prédicat de type $[\langle t_1, \dots, t_n \rangle] = [t_1] \to \cdots \to [t_n] \to o$.

Définition 4 (Chauffage)**.** Le chauffage est l'application maximale des règles $\parr$,$\perp$ et $\forall$. On note $P^{\uparrow}$ le chauffage de la formule P.

Définition 5 (Classes $\mathbb{E}$ et $\mathbb{D}$)**.** Pour un processus Q, on définit $\mathbb{E}(Q)$ comme la clôture de [2] $\{\lbag [Q] \rbag\}$ par chauffage et contraction et instantiation des existentiels. On note $\mathbb{D}$ la clôture des encodages de définitions par déréliction et instantiation des existentiels, et chauffage.

Nous retrouvons maintenant des lemmes préliminaires similaires aux lemmes 1 et 2, dont les démonstrations ne posent pas de problème.

Lemme 3. *Il n'y a pas de dérivation close pour les formules-processus :*

$$\forall P \,.\, \nvdash [P]^{\uparrow}$$

Lemme 4. *Il n'y a pas de dérivation close quand deux motifs sont libres :*

$$\forall (a_i)_{i \leq n} \,.\, \forall (b_j)_{j \leq m} \,.\, \forall P \,.\, \nvdash {a_1}^{\perp} \otimes \cdots \otimes {a_n}^{\perp}, {b_1}^{\perp} \otimes \cdots \otimes {b_m}^{\perp}, [P]^{\uparrow}$$

Théorème 2. *Comme précédemment, toute réduction correspond à une dérivation ouverte :*

$$P \longrightarrow^{*} Q \;\Rightarrow\; \dfrac{\vdash [Q]^{\uparrow}}{\vdash [P]}$$

La réciproque est compliquée par l'apparition de formules parasites :

$$
\dfrac{\vdash \Gamma}{\vdash [P]} \;\Rightarrow\; \begin{cases} \exists Q \,.\, \exists Q' \in \mathbb{E}(Q) \,.\, \exists D \in \mathbb{D} \,.\, \Gamma = Q' \setminus D \\[2ex] \forall Q \,.\, \forall Q' \in \mathbb{E}(Q) \,.\, \forall D \in \mathbb{D} \,.\, \Gamma = Q' \setminus D \quad \Rightarrow \quad \dfrac{\vdash [Q]^{\uparrow}}{\vdash [P]} \wedge P \longrightarrow^{*} Q \end{cases}
$$

Pourquoi ces complications ? La difficulté vient ici de l'inadéquation entre les notions de réplication du Join et de la logique linéaire. Pour obtenir Q, nous devons d'abord supprimer de Γ des définitions instanciées. Ensuite, il faudra ajouter des définitions perdues par affaiblissement, ou déréliction sans contraction préalable.

[1] Par convention, o est le type des formules.

[2] $\lbag$ et $\rbag$ dénotent un multi-ensemble.

La technique utilisée pour encoder le passage de valeur pose nettement moins de problèmes.

Q **n'est pas unique.** Afin de préciser l'énoncé, on se demande s'il n'existe pas un Q minimal, voire unique. C'est faux. Considérons en effet le processus suivant :

$$P = \textbf{let def}\,(a \rhd \textbf{let def}\, b \rhd b \,\textbf{in}\, a) \wedge (a \rhd \textbf{let def}\, b \rhd b \,|\, b \,\textbf{in}\, a)\,\textbf{in}\, a$$

Toute réduction de P ne fait que changer l'ensemble des définitions. A chaque fois on crée un nouveau canal (un b frais) et on ajoute une définition inutile $b \rhd b$, ou $b \rhd b \,|\, b$. Il y a deux réductions incomparables dont les ensembles de définitions sont disjoints : $b \rhd b \vdash a$ et $b \rhd b \,|\, b \vdash a$. Ces deux processus fournissent des témoins incomparables pour la même dérivation ouverte de conclusion P et feuille impropre a.

Démonstration. La première direction ne pose pas problème, et les quelques exemples fournis auront déja convaincu le lecteur. On s'attache à l'autre direction, en se donnant une dérivation 1-ouverte de $[P]$ vers Γ.

Nous allons construire un processus Q et un multi-ensemble de formules D. Pour cela, transformons notre dérivation en respectant la stratégie suivantes : la règle $?W$ n'est autorisée qu'avant la règle initiale ; les règles $?D$ et $\exists I$ sont toujours appliquées consécutivement et sur la même formule ; la règle $?C$ n'est utilisée que si nécessaire : avant un $?D$, ou un $\otimes$ qui envoie une formule répliquée dans chaque sous dérivation. Je ne détaillerai pas la transformation, cette politique "paresseuse" étant simple à mettre en place. Cette gestion des exponentielles est en fait proche de certaines présentations équivalentes [Mil96] de la logique linéaire où les séquents sont explicitement séparés en hypothèses linéaire et hypothèses répliquées.

On obtient une dérivation 1-ouverte de $[P]$ vers Γ', ce dernier ne contenant aucune définition affaiblie ou instanciée. Autrement dit, toutes les définitions apparaissent sous leur forme générale $?(\exists\overline{x}\ .\ \cdots \otimes \cdots)$. Ce multi-ensemble est alors l'encodage d'un processus, Q. Γ' s'obtient à partir de Γ par chauffage, ajout de définitions, et suppressions de définitions spécialisées (affaiblissement ou instantiation). Les ajouts sont placés dans $Q' \in \mathbb{E}(Q)$, les suppressions dans $D \in \mathbb{D}$, et on a bien $\Gamma = Q' \setminus D$.

Ayant traité ce problème de réplication, il reste à réduire les sous-dérivations gauches issues de $\otimes$. Le raisonnement est le même que pour le théorème 1, en utilisant les lemmes 3 et 4. On obtient une dérivation dans laquelle les branches gauches sont des preuves de la forme $\vdash J(\overline{x}), a_1(\overline{x}_1), \ldots, a_n(\overline{x}_n)$, consistant bien à vérifier qu'une définition peut être appliquée. $\square$

Si l'on admet des définitions de la forme $a(c) \rhd P$ où c est une constante, on ne peut plus différencier une définition d'une définition instanciée. Dès lors on ne peut "corriger" comme précédemment la feuille impropre en enlevant les définitions instanciées trop tôt. On est forcé d'admettre la réduction suivante :

$$a(x) \vdash P \quad \rightarrow \quad a(c) \vdash P$$

En réalité, cela ne poserait pas de problème en Join-calcul, pour une mauvaise raison. Les définitions étant toutes répliquées, la modalité ? les encadrant permet de distinguer une définition d'une définition en cours d'utilisation (contraction, déréliction, instantiation de l'existentiel). Dans un encodage du π-calcul, cela ne fonctionnerait plus.

3.4 Séquentialité, synchronisation

On a vu que la logique linéaire multiplicative rendait compte de la dynamique des algèbre de processus asynchrones. Cependant, rien ne semble pouvoir correspondre aux calculs synchrones.

Dans *A purely logical account of sequentiality* [Bru02], Paola Bruscoli encode le CCS fini synchrone dans le calcul des structures [Gug02], doté d'un connecteur de synchronisation. Elle obtient ainsi un résultat similaire au notre, les traces de CCS étant équivalentes aux dérivations ouvertes dans la logique. Nous introduisons un opérateur de synchronisation dans le calcul de séquents MLL en préservant l'élimination des coupures et des *init* non atomiques. On obtient ainsi un résultat analogue à celui de [Bru02], dans un cadre plus élémentaire.

3.4.1 Logique

On travaille toujours dans le fragment multiplicatif de la logique linéaire [Gir87], défini Figure 1. Afin d'encoder le synchronisme, on propose deux nouveaux connecteurs :

$$\frac{\vdash \Gamma, P, Q}{\vdash \Gamma, \bullet P, \bullet Q}\ \bullet \qquad \frac{\vdash \Gamma, P, Q}{\vdash \Gamma, \bullet P, \bigcirc Q}\ \bullet\!\bigcirc$$

Ces deux règles expriment la synchronisation dans sa plus simple forme. Le premier cas est symétrique. Dans le second, on distingue deux rôles, un émetteur et un recepteur.

Le comportement de ces nouveaux connecteurs n'est pas *canonique*. En effet, il n'est pas défini en isolation, mais en interaction avec un autre connecteur. De ce fait, on peut dupliquer les connecteurs en une famille indexée de connecteurs de synchronisations suivant les mêmes règles, et rien ne pourra les distinguer dans la logique. En faisant cela, nous obtenons la notion de canal.

Tout en dupliquant ces primitives de synchronisation, on obscurcit leur différence. On se donne un ensemble d'actions, et une involution $a \mapsto \overline{a}$. Pour chaque action a on se donne un connecteur unaire $a.$, et l'on équipe la logique obtenue de la règle suivante :

$$\frac{\vdash \Gamma, P, Q}{\vdash \Gamma, a.P, \overline{a}.Q}$$

Lorsque $a = \overline{a}$ il s'agit d'une instance de la règle $\bullet$, sinon c'est une instance de $\bullet\!\bigcirc$.

On nomme MSLL la logique ainsi obtenue, *Multiplicative Synchronous Linear Logic*. Nous montrons que MSLL jouit toujours d'élimination des coupures, et qu'on peut réduire l'utilisation de la règle initiale aux instances atomiques. Après cela, nous verrons que le CCS fini synchrone s'encode dans cette logique.

Théorème 3. *Dans toute dérivation, les instances non atomiques de la règle initiale peuvent être éliminées.*

Démonstration. Cette propriété est vraie pour MLL, vérifions la pour le nouveau connecteur. On repousse simplement une instance non-atomique vers une instance plus petite comme suit :

$$\frac{}{\vdash a.P, \overline{a}.P^{\perp}}\ init \quad \longrightarrow \quad \frac{\dfrac{}{\vdash P, P^{\perp}}\ init}{\vdash a.P, \overline{a}.P^{\perp}}\ sync$$

□

$$\begin{array}{|cc|}\hline A^{\perp\perp} = A & \perp^\perp = \mathbf{1} \\ (A \parr B)^\perp = A^\perp \otimes B^\perp & (A.B)^\perp = A^\perp.B^\perp \\ \hline \end{array}$$

$$\begin{array}{|cc|}\hline \dfrac{\vdash \Gamma}{\vdash \Gamma, \perp}\ \perp & \dfrac{}{\vdash \mathbf{1}}\ \mathbf{1} \\[2ex] \dfrac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ \otimes & \dfrac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \\[2ex] \dfrac{}{\vdash A, A^\perp}\ init & \dfrac{\vdash \Gamma, P, Q}{\vdash \Gamma, a.P, \overline{a}.Q}\ sync \\ \hline \end{array}$$

FIG. 3 – Multiplicative Synchronous Linear Logic

Théorème 4. *La coupure est admissible dans MSLL.*

Démonstration. Le cas délicat apparait lorsque une synchronisation a lieu sur la formule introduite par coupure. Considérons une dérivation de cette forme, et repoussons la coupure.

$$\dfrac{\dfrac{\dfrac{\Theta_1}{\vdash \Gamma_1, P, Q}}{\vdash \Gamma_1, a.P, \overline{a}.Q}\ sync \qquad \dfrac{\Theta_2}{\vdash \Gamma_2, a.Q^\perp}}{\vdash \Gamma_1, \Gamma_2, a.P}\ cut$$

Considérons maintenant la première règle utilisée dans Θ_2 :

- Si ce n'est pas une synchronisation sur $a.Q^\perp$, la dérivation Θ_2 a la forme générale suivante[3] :

$$\dfrac{\Pi_1 \quad \ldots \quad \Pi_n \quad \dfrac{\Pi'_1}{\vdash \Delta_1, a.Q^\perp} \quad \ldots \quad \dfrac{\Pi'_m}{\vdash \Delta_m, a.Q^\perp}}{\vdash \Gamma_2, a.Q^\perp}$$

  On en obtient une nouvelle preuve de $\vdash \Gamma_1, \Gamma_2, a.P$, avec une coupure plus profonde :

$$\dfrac{\Pi_1 \quad \ldots \quad \Pi_n \quad \ldots \quad \dfrac{\dfrac{\dfrac{\Theta_1}{\vdash \Gamma_1, P, Q}}{\vdash \Gamma_1, a.P, \overline{a}.Q}\ sync \qquad \dfrac{\Pi'_i}{\vdash \Delta_i, a.Q^\perp}}{\vdash \Delta_i, \Gamma_1, a.P}\ cut \quad \ldots}{\vdash \Gamma_2, \Gamma_1, a.P}$$

- Dans le cas où Θ_2 commence par une synchronisation impliquant $a.Q^\perp$, on a $\Gamma_2 = \Delta, \overline{a}.R$ et Θ_2 vaut :

$$\dfrac{\dfrac{\Theta}{\vdash \Delta, R, Q^\perp}}{\vdash \Delta, \overline{a}.R, a.Q^\perp}\ sync$$

[3] La présentation est un peu lourde, mais on notera que cette forme générale vaut aussi pour les additifs, démontrant ainsi que la synchronisation peut être ajoutée à la logique linéaire dans son intégralité sans compromettre l'élimination des coupures.

On ajoute Γ_1, et on transforme cela en une preuve de $\Gamma_1, \Gamma_2, a.P$ avec une coupure repoussée :

$$\dfrac{\dfrac{\dfrac{\Theta_1}{\vdash \Gamma_1, P, Q} \quad \dfrac{\Theta}{\vdash \Delta, Q^{\perp}, R}}{\vdash \Gamma_1, \Delta, R, P}\ cut}{\vdash \Gamma_1, \Delta, \overline{a}.R, a.P}\ sync$$

□

Il n'y a *a priori* aucune raison de structurer l'ensemble des actions par une involution. On peut concevoir la logique de façon plus générale, en munissant les actions d'une relation binaire $\mathcal{R}$, et en changeant la règle d'inférence :

$$\frac{\vdash \Gamma, P, Q}{\vdash \Gamma, a.P, b.Q}\ sync \quad if\ a\mathcal{R}b$$

Cependant, l'élimination des coupures impose $\forall a, b, c\ .\ a\mathcal{R}b \Rightarrow c\mathcal{R}\overline{b} \Rightarrow c\mathcal{R}a$. En fait, $\mathcal{R}$ est l'union de cliques de de graphes bipartis, on n'a donc rien gagné par rapport à l'involution.

Cette étude logique d'un nouveau connecteur non canonique est similaire à [DJS93]. Les auteurs de cet article étudient la duplication des exponentielles en une famille équipée d'une relation d'accessibilité régissant les interactions possibles entre les instances des modalités. Ils établissent les conditions nécessaires à la préservation de l'élimination des coupures et des règles initiales non atomiques : respectivement transitivité et reflexivité de la relation d'accessibilité.

3.4.2 Calcul

Nous donnons maintenant une interpretation calculatoire de notre nouveau connecteur, en encodant le CCS fini synchrone [Mil80]. Les processus sont définis selon la grammaire suivante :

$$P ::= \quad 0 \quad | \quad \overline{a}.P \quad | \quad a.P \quad | \quad (P\,|\,P)$$

Ils sont identifiés, comme précédemment, modulo associativité et commutativité de la composition parallèle, et neutralité de 0. La seule règle de réduction est :

$$\overline{a}.P\,|\,a.Q\,|\,C \longrightarrow P\,|\,Q\,|\,C$$

On encode les processus dans les formules de MSLL :

$$\begin{array}{rcl} [0] & = & \perp \\ [a.P] & = & a.[P] \\ [\overline{a}.P] & = & \overline{a}.[P] \\ [P\,|\,Q] & = & [P] \parr [Q] \end{array}$$

Théorème 5 (Reduction & Deduction)**.** *La réduction de CCS synchrone fini correspond exactement à la déduction dans MSLL :*

$$\forall P \,.\, \forall Q \,.\quad P \to^* Q \quad\Rightarrow\quad \frac{\vdash [Q]^\uparrow}{\vdash [P]} \tag{1}$$

$$\forall P \,.\quad \frac{\vdash \Gamma}{\vdash [P]} \quad\Rightarrow\quad \begin{cases} \exists! Q \,.\, \Gamma^\uparrow = [Q]^\uparrow \\ \forall Q \,.\, \Gamma^\uparrow = [Q]^\uparrow \Rightarrow P \to^* Q \end{cases} \tag{2}$$

Ce théorème est une trivialité, les dérivations ne comportant pas de branchement quand le but est un encodage de processus. Par contre, une extension intéressante peut-être d'utiliser un seul nouveau connecteur binaire, ramenant les actions dans la logique en les représentant par des atomes comme on le fait pour les calculs asynchrones.

$$\frac{\vdash \Gamma, A, B \quad \vdash \Delta, P, Q}{\vdash \Gamma, \Delta, A.P, B.Q}$$

Cela permet notamment d'encoder des calculs synchrones avec passage de valeur, par une technique similaire à celle utilisée pour le Join-calcul. Je ne détaille pas cet encodage supplémentaire, la preuve étant encore une fois très similaire. On vérifie aussi aisément que MLL étendue avec ce nouveau connecteur de synchronisation binaire vérifie toujours l'élimination des coupures et des *init* non atomiques. La preuve du théorème s'obtient à partir de 4, qui en est en fait une version simplifiée.

4 Application à la sémantique

Nous montrons maintenant comment nos résultats peuvent être appliqués à la vérification du comportement de processus. Notre approche est fondée sur la sémantique par tests (*testing semantics*). Nous nous placons sauf mention contraire dans le cadre du CCS asynchrone fini. En effet, le CCS synchrone repose sur une logique étendue et moins étudiée (pas de sémantique des phases), et le Join-calcul se prête mal à la sémantique par tests.

Ce dernier point mérite une explication. Une façon naturelle d'analyser le comportement d'un processus CCS est de le plonger dans un environnement. On vérifie ainsi qu'un processus qui demande l'heure va terminer, quand on le confronte à un serveur qui donne l'heure. En Join-calcul, une réception ne peut se faire que dans une définition, qui est accompagnée d'un extrusion de contexte. Par conséquent, Q ne peut recevoir que sur un nom frais. Pour tester si P émet sur les noms libres a et b, on devra considérer le processus **let def** $a \,|\, b \triangleright w$ **in** P et non pas une simple composition parallèle. Ainsi les testeurs sont des contextes. De plus leur forme est très spécifique du point de vue logique, comme les processus eux-mêmes. Au contraire, la structure des formules pour aCCS est très peu contrainte.

Bien sûr, se restreindre à aCCS est faible, car ce calcul est extrêmement simple. La terminaison et l'équivalence sont décidable. Cependant, notre approche devrait s'étendre à un calcul avec réplication, et je l'espère avec passage de valeur, ce qui est plus délicat.

Retournons donc au CCS asynchrone fini, et considérons de plus la logique linéaire comportant les additifs : MALL. Nous allons donner sens à tous les connecteurs logiques dans le cadre des testeurs.

$$\top^\perp = \mathbf{0} \qquad \frac{}{\vdash \Gamma, \top}\ \top$$

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

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

FIG. 4 – Le fragment additif de la logique linéaire

La définition des testeurs est une paraphrase de la définition de MALL. On notera cependant l'interprétation de $\otimes$ quand il n'est plus restreint à un membre atomique : on obtient alors une séparation au sens de la logique spatiale. La constante **1** sert à tester des terminaisons. **0** est un *deadlock*, une impossibilité de succès. $\top$ est le canal spécial w utilisé dans la définition des testeurs de CCS. Les additifs & et $\oplus$ sont lus naturellement comme des connecteurs logiques : P satisfait $\Gamma_1\&\Gamma_2$ s'il satisfait séparemment les deux.

On interprète ainsi toute la logique MALL en terme de processus et testeurs, et du même coup on rapproche ces deux notions — ce qui est peu étonnant.

Définition 6. Un processus P satisfaira un test Γ si $\vdash [P], \Gamma$ et on appelera comportement de P l'ensemble des tests qu'il satisfait, noté $[\![P]\!]$.

Il convient de remarquer que satisfaire un test n'est pas un comportement nécessaire, mais seulement une exécution possible. Il s'agit de la distinction entre *must-testing* et *may-testing*.

Les nombreux opérateurs disponibles permettent d'écrire des tests très expressifs. Cependant la plupart n'apportent rien quand on considère les tests satisfaits dans leur ensemble. On sait par exemple que la sémantique par trace, qui à un processus associe l'ensemble des traces qu'il peut réaliser, est équivalente à la sémantique par testeurs, n'utilisant donc que la constante $\top$ en plus des constructions de processus usuels.

Par exemple, en CCS synchrone, vérifions que P peut recevoir sur b puis émettre sur a :

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash c.\perp, \perp, \top}}{\vdash c.\perp, \overline{a}.\perp, a.\top}}{\vdash b.c.\perp, \overline{b}.\overline{a}.\perp, a.\top}}{\vdash b.c.\perp, b.\overline{b}.\overline{a}.\perp, \overline{b}.a.\top}}{\vdash b.c.\perp \parr b.\overline{b}.\overline{a}.\perp, \overline{b}.a.\top}$$

La notion de comportement définie ici n'est pas exactement celle utilisée en concurrence. En effet, il y a dans l'ensemble des testeurs des formules ne correspondant pas à un encodage de testeur au sens de la concurrence, notamment puisque l'utilisation du tenseur n'est plus restreinte à un membre gauche atomique. Par conséquent le quotientage par équivalence comportementale va être plus discriminant qu'en concurrence. Autrement dit on a seulement :

$$[\![P]\!] \subset [\![Q]\!] \Rightarrow Q \text{ simule } P$$

Cependant cette notion a l'avantage d'être élémentaire en logique, et permet d'utiliser les connaissances en recherche de preuve en logique linéaire pour prouver des inclusions de comportements, au sens de la concurrence cette fois.

Proposition 1. $[P] \vdash [Q] \Leftrightarrow [\![P]\!] \subset [\![Q]\!]$

Des variations sur cette observation nous permettent de déduire immédiatemment certaines propriétés fondamentales du calcul à partir de celles de la logique.

Proposition 2. $P \to^* Q \Rightarrow [\![Q]\!] \subset [\![P]\!]$

Proposition 3. $[P] \equiv [Q] \Leftrightarrow [\![P]\!] = [\![Q]\!]$

Proposition 4. $[\![a^\perp \otimes b^\perp \otimes P]\!] = [\![b^\perp \otimes a^\perp \otimes P]\!]$

En définissant la notion de test, nous sommes retournés dans le domaine bien connu des dérivations fermées, des preuves à part entière. Cela mène à une application intéressante. Okada et Terui [OT96] remarquent en effet que la sémantique des phases [Gir87] permet de vérifier des propriétés négatives. [FRS98] utilise ces résultats pour raisonner sur le fameux problème des philosophes. L'existence d'une dérivation de Γ correspond à la validité de la formule dans toute interprétation, par conséquent l'existence d'une interprétation invalide permet d'affirmer que Γ n'est pas dérivable. Plus précisément :

$$\exists P \,.\, \exists \eta \,.\, \eta(\Gamma^\perp) \not\subset \eta(P) \quad \Rightarrow \quad \Gamma \nvdash P$$

La transformation d'un problème universel en un existentiel est très intéressante. Mais cette technique reste limitée. Il est en effet très difficile de construire un contre-exemple en sémantique des phases. Cela mène Okada et Terui à définir une sémantique dite naïve, plus simple, pour un fragment de la logique linéaire. Ce fragment permet d'encoder CCS asynchrone fini sans passage de valeurs, et autorise l'utilisation de la réplication, mais uniquement au niveau le plus extérieur — $!a.(P|Q)$ est autorisé mais pas $a.(P|!Q)$.

5 Conclusion

Nous avons vu qu'il existe une analogie simple et complète entre la logique linéaire et de nombreuses algèbres de processus. Nous étendons cette analogie de façon uniforme à un calcul synchrone, au passage de valeur, et proposons notamment un encodage du Join-calcul.

Une direction pour de futurs travaux est ouverte par l'utilisation de connecteurs non canoniques pour encoder le synchronisme, et de façon plus générale structurer le calcul/la preuve. Par exemple, les modalités de la logique linéaire ne vont pas sans évoquer une certaine notion de localité. L'idée générale est que A ne peut interagir qu'avec les formules de Γ dans le séquent $\vdash !\Gamma, ?A, \Delta$. En dupliquant ensuite les modalités comme on fait pour la synchronisation, on fait apparaitre plusieurs compartiments, soit une notion de localité. D'un point de vue purement logique, la duplication en une famille de modalités a déja été étudiée dans [DJS93]. L'application à la programmation logique linéaire n'est cependant pas pleinement satisfaisante dans la mesure où elle n'exprime finalement que quelques aspects restreints de la notion de localité. Notamment, je ne vois à l'heure actuelle aucun moyen d'exprimer de la mobilité.

Les applications de nos résultats sont encore assez limitées. La logique linéaire peut être vue comme un langage formel pour raisonner sur la concurrence. Ainsi un environnement de preuve équipé pour la logique linéaire pourrait être utilisé pour raisonner sur des calculs de la concurrence et les exécuter. C'est un premier pas, mais on peut aussi attendre des conséquences plus profondes d'une telle analogie. Le raffinement des énoncés et l'encodage de nouvelles notions que nous apportons ici constitueront peut-être un pas vers une meilleure compréhension.

Table des matières

Références

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3) :297–347, 1992.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96 :217–248, 1992.

[Bef05] Emmanuel Beffara. A concurrent model for linear logic. In *Proceedings of the Mathematical Foundations of Programming Semantics (MFPS)*, 2005. To appear in ENTCS.

[Bru02] Paola Bruscoli. A purely logical account of sequentiality in proof search. In Peter J. Stuckey, editor, *Logic Programming, 18th International Conference*, volume 2401 of *LNAI*, pages 302–316. Springer-Verlag, 2002.

[DJS93] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. The structure of exponentials : Uncovering the dynamics of linear logic proofs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 159–171. Springer, 1993.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL '96 : The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996.

[FRS98] François Fages, Paul Ruet, and Sylvain Soliman. Phase semantics and verification of concurrent constraint programs. In Vaughan Pratt, editor, *Symposium on Logic in Computer Science*. IEEE, July 1998.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50 :1–102, 1987.

[Gug02] Alessio Guglielmi. A system of interaction and structure. Accepted by ACM Transactions on Computational Logic, 2002.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[KY93] Naoki Kobayashi and Akinori Yonezawa. ACL - a concurrent linear logic programming paradigm. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 279–294. MIT Press, October 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, New York, NY, 1980.

[Mil93] Dale Miller. The π-calculus as a theory in linear logic : Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265, Bologna, Italy, 1993. Springer-Verlag.

[Mil96] Dale Miller. Forum : A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1) :201–232, September 1996.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51 :125–157, 1991.

[OT96] Mitsuhiro Okada and Kazushige Terui. Semantic characterizations for reachability and trace equivalence in a linear logic-based process calculus (preliminary report). Technical report, Dept. of Philosophy, Keio University, 1996.