THÈSE

présentée à

L'UNIVERSITÉ d'ORLÉANS

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ D'ORLÉANS

Discipline : INFORMATIQUE

par

Jean-Hugues RÉTY

Titre de la thèse :

Langages Concurrents avec Contraintes : Communication par Messages et Distribution

Soutenue le 6 janvier 1997
devant la commission d'examen composée de :

MM.	J.L. Imbert	....	Président
	F. Fages	....	Rapporteur
	D. Gilbert	....	Rapporteur
	F. Benhamou	....	Examinateur
	P. Codognet	....	Examinateur
	G. Ferrand	....	Examinateur

Table des matières

Résumé

Issus des Langages Logiques Concurrents et de la Programmation Logique avec Contraintes, les langages Concurrents avec Contraintes (CC) forment un paradigme de calcul par informations partielles très attractif. La notion de *store* global dont l'origine se trouve dans la Programmation Logique – la substitution calculée d'un noeud de l'arbre SLD peut être représentée par un ensemble d'équations sur les termes (contraintes sur le domaine de Herbrand) – est un obstacle évident à la distribution de ce type de langages. Bien que certains modèles de programmation distribuée soient basés sur une mémoire partagée, la résolution de contraintes distribuée étant un problème difficile et mal maîtrisé à l'heure actuelle, il semble malaisé d'implanter un calcul par accumulation de contraintes dans un store global de manière distribuée. D'autre part, la plupart des langages de programmation distribuée proposent un mécanisme de calcul local (impératif, avec communication par mémoire locale) et un mécanisme de communication inter sites (communication par messages).

Nous proposons un modèle de calcul distribué par contraintes dont le mécanisme de calcul local est un langage CC et dont la communication inter sites s'effectue par messages. Les canaux de communication sont des variables (logiques) du système de contraintes, aussi bien au niveau local (communication *ask/tell* des CC) qu'au niveau global (communication par messages). Les contraintes permettent une modélisation logique de la communication par messages, cette dernière s'intégrant de manière très homogène dans les CC. Nous appelons ce modèle de calcul MCC (Message-based CC).

Le modèle MCC est très général et laisse une grande liberté d'utilisation des deux modes de communication. Nous définissons une notion de *distribuabilité* qui capture le fait que des processus en composition parallèle peuvent s'exécuter indépendamment les uns des autres sauf en ce qui concerne la communication par messages. L'idée intuitive sous-jacente est que dans ce cas, les *ask* et les

tell d'un processus ne portant que sur des variables qui lui sont locales, une implantation distribuée est facilitée par l'absence de résolution distribuée de contraintes. Cette notion de distribuabilité est non décidable. Nous définissons un sous langage des MCC appelé DCC (Distributed CC) ainsi qu'une notion de *site* sur les agents DCC. Nous prouvons qu'une composition parallèle de *sites* est distribuable. Les DCC offrent un pouvoir d'expression important ; notons d'ailleurs qu'un programme DCC qui n'utilise pas la communication par messages est un programme CC. Nous donnons des exemples issus des domaines de la concurrence et de la distribution.

Nous définissons une relation de bisimilarité sur les DCC et donnons un encodage du π-calcul dans les DCC qui est prouvé être complètement adéquat.

Chapitre 1

Introduction

L'histoire de la théorie des langages de programmation est une continuelle élévation vers l'abstraction : abstraction des contingences liées à la machine, en ce qu'elle est objet physique, mécanique, élévation vers un mode d'utilisation de celle-ci plus proche de nos modes de raisonnement. L'ordinateur, au sens physique du terme – *hardware* – n'est qu'une machine à la fonctionnalité très faible. Il ne connait qu'un nombre assez restreint d'opérations de base – instructions machine –, lecture/écriture en mémoire, opérations arithmétiques, lecture/écriture sur des périphériques... Son efficacité, son vaste champ d'utilisation, lui proviennent de l'immense potentiel d'effectuer des tâches diverses que représente la combinaison complexe de ces opérations simples, et de sa prodigieuse rapidité d'exécution.

Les premiers ordinateurs étaient programmés directement en *langage machine*, en binaire puis en hexadécimal. L'assembleur fut inventé alors afin d'alléger quelque peu le fastidieux travail d'introduction d'une suite d'instructions. Bien que représentant un progrès, l'assembleur n'est cependant que paraphrases, traduction quasi immédiate entre l'hermétique de l'hexadécimal et le un peu plus intuitif des mnémoniques ; ce mot, *mnémonique*, est d'ailleurs ici significatif. Finalement, cela n'est que codage.

L'élaboration de la résolution d'un problème par des moyens informatiques est essentiellement dirigé par le mode de fonctionnement de la machine elle-même. Ce dernier, qui ne peut s'analyser que d'un point de vue opérationnel, est très peu naturel à l'homme. Il s'ensuit que la résolution d'un problème comporte, en particulier, une phase d'analyse qui s'effectue à l'aide d'outils mathématiques, et une phase d'implantation. Implanter en assembleur est un travail très lourd et très long.

Afin de simplifier ce travail et réduire son coût, on a alors eu l'idée d'offrir au programmeur des langages de plus haut niveau, plus abstraits de la machine : ce que l'on appelle les *langages évolués*. Ils permettent la construction de programmes plus rapides à écrire, moins longs en taille de code, et plus faciles à lire. Parmi les langages évolués, la classe des *langages impératifs* regroupe la pluspart des langages de programmation connus du grand public (Pascal, C, ...). Une caractéristique essentielle en est la notion de *variable impérative*. Celle-ci est une abstraction de la notion physique de case mémoire : elle en est abstraite dans le sens où sa taille n'est pas fixée par la machine et où son contenu possède souvent un certain type (entier, booleen,...). Cependant, elle y colle encore de très près dans la mesure où son mode de fonctionnement est celui de la case mémoire, les opérations basiques dont nous disposons étant alors la lecture et l'écriture. En conséquence, sa valeur est suceptible d'être modifiée au cours de l'exécution du programme. Cette dernière propriété s'avère être un obstacle à une compréhension non strictement opérationnelle des programmes. La programmation impérative à donné naissance à des langages puissants et très généraux qui sont de loin les plus utilisé aujourd'hui mais ceux-ci restent proche du fonctionnement de l'ordinateur ; leurs modèles de calculs sont d'ailleurs basés sur les *machines de Turing* ([Her88]) qui forment un modèle de fonctionnement de l'ordinateur en tant que machine. La distance importante entre la modélisation d'un problème et le langage de programmation rend la tâche d'implantation longue et fastidieuse.

La recherche vers des langages de programmation de plus haut niveau apparaît d'un intérêt évident. L'objectif est de rapprocher la programmation de la modélisation des problèmes afin de simplifier le travail d'implantation. Il ne sagit pas de rêver un paradigme qui permette d'exécuter directement cette modélisation (cela semble bien lointain à l'heure actuelle) mais de tendre dans cette direction. Un autre point important concerne la validation et l'analyse de programmes, celles-ci étant très difficile dans le cas des langages impératifs de par leur compréhension presque uniquement opérationnelle. C'est vers cela que se dirige la programmation dite *déclarative*.

1.1 La notion de programmation déclarative

La notion de déclarativité en théorie des langages de programmation n'est pas définie de manière précise et il semble difficile de dégager un réel consensus à son sujet. Nous nous contenterons donc ici de tenter d'en cerner les contours.

Une première approche peut être de dire qu'un langage de programmation est *déclaratif* s'il possède une sémantique déclarative. Cette définition est cependant beaucoup trop générale puisqu'elle englobe à priori les langages impératifs pour lesquels des sémantiques dénotationelles ont été proposées. D'autre part, nous n'avons bien entendu pas beaucoup avancé, il resterait à préciser ce qu'est une sémantique déclarative. Nous proposons plutôt de rephraser cette définition tout en la renforcant quelque peu afin de donner une idée intuitive de sa signification : un langage de programmation est dit *déclaratif* si les programmes écrits dans ce langage peuvent se comprendre de façon déclarative. Déclaratif est ici opposé à opérationnel. Cela signifie qu'il est possible de comprendre un programme sans avoir recours à l'exécution de celui-ci. Nous opposons ainsi le *faire* au *dire*. C'est-à-dire, le *comment cela se construit* à la *description*. L'algorithme d'Euclide, par exemple, est l'énumération d'une suite d'opérations qui prend en entrée deux entiers x et y et renvoie un entier d. Comprendre ce qu'il calcule (le plus grand diviseur commun d de x et y) nécessite de "dérouler" son exécution. Au contraire, l'énoncé *d est le plus grand entier tel que d divise x et d divise y* décrit directement – de manière déclarative – ce que représente d par rapport à x et y. La sémantique d'un langage impératif n'est rien d'autre (ou guère plus) que la manière dont il s'exécute. Il manque à ce type de langages une sémantique de plus haut niveau (et leurs sémantiques dénotationnelles ne sont pas déclaratives au sens décrit ci-dessus). Il en résulte, en particulier, que les programmes écrits dans ce type de langages sont difficiles à analyser et à valider. Afin d'être sûr de ce qui est effectivement calculé par le programme C qu'il a écrit, un programmeur se base essentiellement sur son expérience du langage, sur la connaissance intuitive qu'il en a. Les méthodes de validation de programmes basées sur la logique de Hoare restent ici assez limitées. Nous espérons de l'approche déclarative, entre autre, de permettre des méthodes de preuves beaucoup plus efficientes par ce que basées sur une compréhension non exclusivement opérationnelle du langage.

Les deux grandes approches de la programmation déclarative sont la programmation en logique et la programmation fonctionnelle. Nous restreindrons notre discussion à la première, celle-ci étant à l'origine des langages concurrents avec contraintes qui font l'objet de cette thèse.

1.1.1 La Programmation en Logique

La Programmation en Logique (PL) ([Doe94]) est née au début des années 70 des travaux de Colmérauer et de Kowalski. L'idée fondamentale est de considérer la logique comme un langage de programmation. Celle-ci n'avait été utilisée jusqu'alors en informatique que comme un langage de spécification. Cependant, la logique possède une interprétation procédurale qui la rend très efficace en tant que langage de programmation. La sémantique opérationnelle standard de la programmation logique est la résolution SLD. Celle-ci restreint le langage aux clauses de Horn, ce qui exclut toute information négative. La question de la négation en programmation logique est d'ailleurs un problème difficile et non encore résolu, aussi bien d'un point de vue sémantique qu'opérationnel (cf. [Rét95a]).

La programation en logique bénéficie d'une sémantique logique et d'une sémantique de point fixe bien maitrisées. Selon l'équation de Kowalski : programmation en logique = logique + contrôle, un programme est une théorie logique associée à un certain mécanisme permettant de diriger la recherche dans l'espace de calcul (arbre de recherche). Ce mécanisme est simplement assuré dans les langages de type PROLOG par la prise en compte lors de l'exécution des atomes et des clauses dans l'ordre ou ils sont écrits dans le programme. Ces langages sont donc séquentiel, la stratégie de recherche étant spécifiée de manière stricte. Cette séquentialité, et plus généralement la nécessité d'assurer un certain contrôle de la stratégie d'exécution (cette remarque déborde d'ailleurs largement le cadre de la PL pour s'appliquer à tout paradigme de programmation), est bien souvent un obstacle à la compréhension déclarative de la programmation.

1.2 La concurrence et la distribution

La concurrence, en théorie des langages de programmation, s'oppose à la séquentialité, et une façon d'exprimer les choses serait de dire que les langages séquentiels spécifient un ordre total d'exécution des processus tandis que les langages concurrents spécifient un ordre partiel. Mais elle est plus que cela : elle signifie non seulement que l'ordre d'exécution des processus (ce mot est ici employé en toute généralité) n'est pas spécifié d'une manière totale mais aussi que certains d'entre eux peuvent être exécutés simultanément – en parallèle – et les sémantiques des langages de programmation concurrents peuvent être classées, entre autre, selon leur degré d'expression de cette simultanéité ; cela allant de l'*interleaving*, qui est l'expression de tous les ordres possibles d'exécution séquentielle, à la *true concurrency*, qui est la définition d'un ordre partiel sur les pas d'exécution (transitions).

Tout langage concurrent possède une notion de *communication*, qui est la manière dont les processus concurrents échangent de l'information, et une notion de *synchronisation*, qui est la manière dont est spécifié cet ordre partiel. Les modèles de la concurrence peuvent se ranger en deux grandes catégories : les modèles à communication par mémoire partagée et ceux à communication par messages ([And91]). La seconde catégorie correspond en général à ce que l'on appelle les modèles de la distribution, bien que certains modèles à mémoire partagée permettent des implantations distribuées assez efficaces.

Nous nous intéresserons dans cette introduction à la la programmation logique concurrente mais surtout au π-calcul ; nous reviendrons sur la première avec les langages concurrents avec contraintes. La programmation logique concurrente est un modèle de programmation à mémoire partagée tandis que le π-calcul est à communication par messages.

1.2.1 Les Langages Logiques Concurrents

La première idée de concurrence en programmation logique est, d'un point de vue historique, la notion de retard (cf. [BFR95]). Elle a donné lieu à toute une génération de langages (Prolog II, mu-Prolog, Gödel,...). La notion de clause gardée, qui est à l'origine des Langages Logiques Concurrents (LLC) ([Sha89, CCRS89] en est une extension. Les LLC ont été dévellopés à partir du début des années 80. Parmis les implantations des LLC, citons Concurrent PROLOG, CP, FCP, et GHC, les trois derniers existant en plusieurs versions selon les mécanismes de synchronisation et de communication utilisés. Le programme de recherche japonais "fifth generation computing", essentiellement basé sur les LLC, a eu une grande influence en créant une dynamique autour de ces langages.

Ces langages sont indéterministes et leurs sémantique pose problème. En particulier, il est difficile de leur définir une sémantique logique à cause de la difficulté à modéliser la synchronisation. Nous retrouverons ce problème avec les langages concurrents avec contraintes qui en sont, pour une part, issus. La notion de contrainte permet cependant une meilleure modélisation de la synchronisation.

1.2.2 Le π-calcul

Parmi les modèles mathématiques de la concurrence, le CCS (Calculus of Communicating Systems) ([Mil89]) apparaît comme un paradigme de calcul destiné à servir de base théorique à la programmation concurrente, au même titre que l'est le λ-calcul pour la programmation fonctionnelle. Il s'agit d'un langage restreint qui reflète des idées opérationnelles simples. Le CCS modélise la communication (synchrone) par messages, et seulement celle-ci : seul l'acte de communication est pris en compte, à l'exclusion d'un éventuel contenu de cette communication. Le grand engouement qu'il succite est dû pour partie à son pouvoir d'expression que l'on peut qualifier d'étonnant compte tenu de la simplicité des mécanismes ainsi modélisés.

Le π-calcul ([MPW89]) est une évolution du CCS ayant pour objet de permettre la reconfiguration dynamique du réseau de communication via l'echange de *noms* de canaux, un canal de communication étant modélisé (c'est aussi le cas dans le CCS) par un nom qui est en quelque sorte son identificateur (notons cependant que les noms jouent en fait à la fois le rôle de constantes et le rôle de variables). Ce sont les seuls objets informatiques manipulés, la différence par rapport au CCS résidant dans le fait que ces noms peuvent être transmis comme contenu d'un message.

On ne peut pas dire que le CCS et le π-calcul soient des modèles de calcul à proprement parler déclaratifs, ou du moins pas par rapport à notre définition précédente de la notion de déclarativité. Leur sémantique est opérationnelle : leur présentation usuelle est basée sur un système de transitions spécifié par un système de règles d'inférences de type SOS[1] ([Plo81]). Cependant, de même que dans le cas du λ-calcul, leurs systèmes de transitions admettent une analyse de haut niveau. La clef de voute en est le concept de *bisimulation*. Il est très important, dans le cadre de la concurrence, de pouvoir statuer de l'équivalence de programmes. Partant de l'idée-mot-d'ordre de Milner selon laquelle observer un processus est communiquer avec lui, deux processus sont équivalents s'ils ont la même *capacité* de communication, s'ils sont interchangeables du point de vue de tout processus communiquant avec eux. L'idée du concept de bisimulation est

[1]Structural Operational Semantics

de capturer cela par l'intermédiaire d'une relation d'équivalence appelée *relation de bisimilarité* ou *équivalence d'observation*, celle-ci (ou du moins certaines restrictions) admettant une caractérisation simple.

Cela nous amène à reconsidérer notre définition précédente de la notion de déclarativité. Si celle-ci semble assez adaptée au cas des langages concurrents présentant un non déterminisme *angélique* (un exemple typique en est donné par la PL dont la sémantique considère l'ensemble de l'espace de recherche), elle l'est moins pour les langages concurrents indéterministes (par exemple les LLC dont l'exécution est réduite au parcours d'une branche de l'espace de recherche). Les premiers sont plus adaptés à la résolution de problèmes transformationnels tandis que les seconds sont souvent orientés vers des applications réactives. La modélisation de problèmes réactifs nécessite cet indéterminisme qui est un obstacle à une compréhension déclarative de la programmation. Etant plus particulièrement intéressés par les seconds dans le cadre de cette thèse, nous affaiblirons donc quelque peu cette volonté de construire des langages de programation déclaratifs tout en nous efforcant de proposer des sémantiques opérationnelles permettant une analyse de haut niveau.

1.3 La notion d'information partielle

Le calcul par informations partielles à pris depuis une dizaine d'années une place importante dans le domaine des langages de programmation avec l'apparition de la programmation par contraintes. L'utilisation des contraintes a débuté en intelligence artificielle et en recherche opérationnelle avec les *problèmes de satisfaction de contraintes*. C'est ensuite à la programmation en logique qu'est dû leur essort dans les années 80. L'idée d'interpréter certains prédicats non plus sur le seul domaine de Herbrand mais sur des domaines pour lesquels on possède des algorithmes de résolution efficaces à naturellement donné lieu à la programmation en logique avec contraintes (PLC).

Les systèmes de contraintes ont d'abord été définis par un langage du premier ordre muni d'une interprétation particulière. Cette formalisation est d'ailleurs utilisée dans de nombreux travaux de PLC. Saraswat à proposé au début des années 90 de définir les systèmes de contraintes en tant que systèmes d'information de Scott ([Sco82]) dans lesquels on retire la condition de cohérence. L'intérêt en est double. D'abord, cette définition ne nécessite pas le recours à un langage du premier ordre, elle est donc plus abstraite. En second, la

PLC devient ainsi de plein droit un modèle de calcul par information partielle. Les systèmes d'information partielle (ou systèmes d'information de Scott) ont été introduits par Dana Scott au début des années 70 dans le but de fournir un modèle au lambda-calcul et ont été largement utilisés en informatique comme domaines pour décrire la sémantique dénotationnelle des langages de programmation. Il peut paraître surprenant de retrouver ceux-ci aux fondements de la notion de contrainte. Il est en tout cas intéressant de constater que plusieurs domaines, partis de bases différentes, peuvent ainsi se rejoindre.

1.3.1 La Programmation en Logique avec Contraintes

La Programmation Logique avec Contraintes (PLC) ([Fag96]) est apparue au début des années 80. Elle a réalisé le paradigme de calcul par accumulation d'informations partielles qui était déjà présent dans la PL mais seulement pour les termes du premier ordre. Elle a apporté à la PL l'efficacité de résolveurs de contraintes spécialisés sur certains domaines particuliers (domaines finis, booléens, équations linéaires sur les réels, ...). Le seul domaine d'interprétation de la PL est le domaine de Herbrand, qui permet d'exprimer une grande variété de structures de données mais engendre de graves problèmes d'efficacité. Considérons par exemple les entiers naturels. Ceux-ci se formalisent en PL par les axiomes de Péano, avec un symbole de constante 0 et un foncteur de succession s. L'entier 1 s´ecrit alors $s(0)$, l'entier 2, $s(s(0))$ et ainsi de suite. Il apparaît immédiatement que la manipulation d'entiers de grande taille selon ce procédé est extrêmement coûteuse. Les langages de type PROLOG ont résolu ce problème par l'introduction d'opérateurs arithmétiques extra logiques. Ceux-ci posent cependant des problèmes d'un point de vue sémantique : leur caractère extra logique empêche leur prise en compte par la sémantique logique. Les contraintes ont apporté une solution beaucoup plus satisfaisante à ce problème en interprétant les programmes non plus sur le seul domaine de Herbrand mais aussi sur d'autres domaines pour lesquels on possède des résolveurs de contraintes efficaces. La PLC est née de cette idée. Sa sémantique bien maîtrisée reprend précisément, étendue aux contraintes, celle de la PL. Son succès est incontestable, y compris dans le milieu industriel, et a apporté un renouveau certain dans le monde de la programmation logique. La recherche concernant les résolveurs de contraintes, en vue d'améliorer leur efficacité et de prendre en compte de nouveaux domaines de contraintes, est à l'heure actuelle très active.

1.3.2 Les langages Concurrents avec Contraintes

Les langages Concurrents avec Contraintes (CC) ([Sar92b, BP94]) sont nés des LLC et de la PLC à la fin des années 80. Les contraintes permettent une formalisation logique de la synchronisation qui a donné lieu à la définition d'une sémantique dénotationnelle pour ces langages mais pose cependant problème quant à la définition d'une sémantique logique ; on peut d'ailleurs remarquer que le mot "logique" est absent de l'appellation "langages concurrents avec contraintes". Les CC forment un modèle de calcul très attractif d'un point de vue théorique mais il n'existe à l'heure actuelle qu'assez peu d'implantations. Citons parmis celles-ci les langages AKL et OZ.

La communication et la synchronisation se font par l'intermédiaire de variables partagées. Les CC, à l'instar de la PL forment un modèle de calcul à mémoire partagée, le calcul procédant par accumulation monotones d'information partielle – contraintes – dans un *store* global. Nous discutterons plus loin des problèmes que cela pose quand à la distribution de ces langages.

Comme pour le π-calcul, les CC reposent sur un ensemble assez restreint d'opérateurs simples de composition : composition parallèle, opérateur de choix local, opérateur de restriction. Ceux-ci sont en fait les mêmes dans ces deux paradigmes de calcul, les CC utilisant un mécanisme de comunication asynchrone basé sur la notion de variable logique et sur les contraintes tandis que le π-calcul modélise la communication par messages. Ces deux paradigmes sont donc finalement très proches. Un autre point, à notre avis, les séparent cependant : c'est le fait que, bien que pouvant encoder une assez large classe de structures de données (le langage Pict ([Pie96]) en est une concrétisation intéressante), le π-calcul ne manipule que des noms de canaux tandis que les CC forment un modèle de calcul par informations partielles. Il semble d'ailleurs difficile d'encoder dans le π-calcul les contraintes dans toute leur généralité (cette question reste cependant ouverte).

1.4 Vers un langage de programmation distribuée basé sur les CC

Nous proposons dans cette thèse un paradigme de calcul par contraintes appelé MCC (Message-based CC) qui intègre des idées issues des langages concurrents avec contraintes (CC) et du π-calcul. Les motivations de nos recherches concernent la modélisation de systèmes distribués dans les langages CC. Nous constatons que certaines caractéristiques essentielles de ces derniers – en particulier la notion de store global – forment un obstacle à cet objectif. Modéliser des systèmes distribués dans les CC nécessiterait d'effectuer de la résolution distribuée de contraintes, ce qui reste à l'heure actuel un problème difficile et mal maîtrisé.

Les recherches dans le domaine de la distribution sont restées essentiellement basées sur la programmation impérative (cf. [And91]) (citons cependant [PN84, BC93] dans le cadre de la PL). La définition d'un langage déclaratif (ou du moins *plus* déclaratif) pour la distribution apparaît donc d'un intérêt immédiat. La construction de modèles de la distribution est un problème auquel il est difficile de répondre dans sa globalité. Les langages de programmation distribuée intègrent le plus souvent deux mécanismes distincts : l'un est un mécanisme de calcul local (impératif dans la plupart des cas), l'autre est un mécanisme de communication et de synchronisation entre sites. Il semble difficile d'unifier ces deux composantes et les modèles existant de la distribution les séparent clairement (le π-calcul étant à ce sujet un cas particulier). Nous pensons qu'un modèle de programmation distribuée basé sur les CC doit différencier communication locale et communication entre sites. Partant du constat de la similitude que présentent les opérateurs de composition des CC et du π-calcul, nous proposons d'introduire dans les CC un notion de communication par messages inspirée de celle du π-calcul (dans sa version asynchrone : il nous paraît naturel, de par l'asynchronisme de la communication $ask/tell$ d'utiliser une communication par messages asynchrone). Il s'avère que les contraintes permettent une formalisation logique de la synchronisation due à cette communication par messages dans un style proche de celui utilisé pour formaliser la communication $ask/tell$, montrant ainsi, s'il en est encore besoin, leur grand intérêt en théorie des langages de programmation.

Les MCC forment un paradigme de calcul très général sur lequel nous définissons une notion de *distribuabilité*. Intuitivement, la composition parallèle $A \parallel B$ de deux agents MCC est distribuable en A et B s'ils ne communiquent que par messages, l'objectif étant d'éviter le recours à la résolution distribuée de contraintes. La distribuabilité est bien sûr indécidable. Nous donnons cependant la caractérisation syntaxique, par l'intermédiaire d'un sous langage des MCC appelé DCC (Distributed CC), d'une large classe de programmes distribuables. Nous définissons pour cela une notion de *site* sur les agents DCC et un résultat important de cette thèse est qu'une composition parallèle de *sites* est distribuable. Nous donnons des exemples classiques de programmes issus de la programmation distribuée et de la programmation concurrente.

Ce sous langage offre un pouvoir d'expression important. Il bénéficie au niveau local de toute la puissance de calcul par contraintes des CC (un agent MCC ou DCC ne faisant pas appel à la communication par messages étant un agent CC) ainsi que de celle du π-calcul dont nous donnons un encodage que nous prouvons être *complètement adéquat*[2]. Les variables des MCC et des DCC sont des variables logiques. Cet encodage est à cet égard très intéressant puisqu'il permet une modélisation des variables du π-calcul (qui sont les canaux de communication) par des variables logiques.

1.5 Contenu de la thèse

Le chapitre 1 est une introduction générale au domaines de recherche abordés et aux objectifs de la thèse.

Le chapitre 2 définit les notions de contrainte et de système de contraintes. Nous donnons d'abord un définition "abstraite" dans laquelle un système de contraintes est un treillis complet muni de certaines structures supplémentaires (opérateurs de cylindrification et éléments diagonaux). Nous définissons l'application d'une substitution (de variables) à une contrainte et prouvons certaines propriétés. Nous donnons ensuite une définition des systèmes de contraintes en terme de systèmes d'information de Scott et étudions les liens existant avec la première définition.

Le chapitre 3 présente les langages Concurrents avec Contraintes. Nous définissons une sémantique opérationnelle par l'intermédiaire d'une congruence structurelle et d'un système de transitions. Ce système de transitions est prouvé être uniformément confluent (la confluence uniforme est une restriction de la confluence forte ([Hue80])). Nous montrons qu'un système de transitions uniformément confluent est uniforme en compléxité, dans le sens où toutes les dérivations terminées sont de même longueur et, s'il existe une dérivation infinie alors il n'existe pas de dérivation terminée finie. Nous étudions la sémantique du non-déterminisme *angélique* et prouvons l'indépendance des observables finis par rapport au *scheduling*.

[2] fully abstract

Le chapitre 4 présente le π-calcul dans sa version asynchrone. Nous définissons sa sémantique opérationnelle dans un style proche de celui utilisé au chapitre 3 par l'intermédiaire d'une congruence structurelle et d'un système de transitions annoté. Nous définissons les bisimilarité fortes et faibles dans une version adaptée à la communication asynchrone issue d'une formalisation récente de Amadio, Castellani et Sangiorgi ([ACS96]).

Le chapitre 5 définit les MCC (Message-based CC). Les MCC forment un paradigme de calcul par contraintes qui intègre langages CC et communication par messages. Nous donnons sa sémantique opérationnelle. Nous définissons une notion de *distribuabilité* qui capture le fait que des processus en composition parallèle peuvent s'exécuter indépendamment les uns des autres sauf en ce qui concerne la communication par messages. Nous proposons un mécanisme de communication du résultat de calculs locaux qui permet de transmettre l'information locale pertinente au fur et à mesure de sa production.

Le chapitre 6 définit les DCC (Distributed CC) qui sont un sous langage des MCC adapté à la programmation distribuée. Nous définissons une notion de *site* sur les agents DCC et nous prouvons qu'une composition parallèle de *sites* est *distribuable* au sens qui est donné à cette notion au chapitre 5. Nous donnons en DCC des exemples issus des domaines de la distribution et de la concurrence.

Le chapitre 7 définit une relation de bisimilarité (dans ses versions fortes et faibles) sur une restriction des DCC. Nous encodons le π-calcul asynchrone dans les DCC et prouvons que cet encodage est *complètement adéquat*. Nous encodons ainsi les variables (qui représentent les canaux de communication) du π-calcul avec des variables logiques et montrons que les DCC ont (au moins) la puissance du π-calul.

Chapitre 2

Les systèmes de contraintes

L'utilisation des systèmes de contraintes à commencé au début des années 70 en Intelligence Artificielle avec les Problèmes de Satisfaction de Contraintes (CSP[1]). Des recherches ont alors été menées afin d'obtenir des algorithmes de résolution de contraintes dont certains sont utilisés aujourd'hui en PLC et dans les CC (e.g. le symplexe pour la résolution des inéquations linéaires sur les réels).

La signification intuitive d'une contrainte – d'où le mot *contrainte* – est de restreindre l'ensemble des valeurs possibles que peuvent prendre certaines variables (les *variables contraintes*). Par exemple, dans le domaine des entiers, la contrainte $2 \leq x \leq 7$ signifie que la valeur de la variable x doit être comprise entre 2 et 7. Autrement dit, $2 \leq x \leq 7$ nous fournit une *information partielle* concernant la valeur de x. Nous discuterons plus loin des liens étroits existant entre les concepts de contrainte et d'information partielle.

Le concept de contrainte a pris une place très importante en programmation en logique et y a amené un renouveau certain. Il a apporté, avec la PLC, une grande efficacité de calcul sur certains domaines (domaines finis, booléens, structures à traits,...) associée à des fondements sémantiques homogènes et clairs. En fait, ces derniers sont l'extension quasi-directe de ceux de la PL – la PL n'est d'ailleurs que la PLC sur le domaine de Herbrand – et on retrouve en particulier les sémantiques de modèles et de points fixes. Notons aussi que [BFB$^+$95] a récemment étendu à la PLC la vision grammaticale de la PL ([DM93, Fer91]), définissant une sémantique en terme d'arbres de preuves.

Maher ([Mah87]) a montré que la synchronisation peut être caractérisée

[1]Constraint Satisfaction Problems

de manière déclarative par l'intermédiaire de la relation de conséquence des systèmes de contraintes. Les travaux de Saraswat ([Sar88, Sar89]) ont repris cette idée et ont débouché sur les langages Concurrents avec Contraintes. Les contraintes se situent aux fondements mêmes de la PLC et des CC. Les domaines sémantiques utilisés sont basés sur les systèmes de contraintes.

Les premières formalisations des systèmes de contraintes utilisées dans le cadre de la PLC ([JL87]) consistent en la donnée d'un langage du premier ordre associé à une interprétation (ou un ensemble d'interprétations), cette dernière étant parfois spécifiée par l'intermédiaire d'une théorie du premier ordre. Les contraintes sont les formules de ce langage ou un sous ensemble de celles-ci. Une relation de conséquence est définie de manière similaire à la relation de conséquence logique en logique du premier ordre.

Saraswat et Rinard ([SR90]) définissent un système de contraintes à partir d'une notion primitive de *contrainte simple* sur laquelle une relation compacte de conséquence est définie. Les systèmes de contraintes peuvent alors être vus comme une extention des domaines de Scott. Nous reviendrons sur ce point dans la section 2.3 de ce chapitre. Un système de contraintes est alors construit sur les ensembles de contraintes simples – les contraintes sont les parties closes par la relation de conséquence – en étendant la relation de conséquence sur ceux-ci. La structure obtenue forme un treillis complet algébrique. L'intéret de cette approche réside – en plus de la relation ainsi effectuée entre systèmes de contraintes et systèmes d'information – dans son aspect constructiviste. Nous nous en abstrayons dans un premier temps pour ne considérer que la structure finale.

2.1 Préliminaires

Nous rappelons dans cette section quelques notions de la théorie des ensembles et de la théorie des treillis qui seront utiles dans ce chapitre ainsi que dans les chapitres suivants.

Soit E un ensemble, nous notons $P(E)$ l'ensemble des parties de E et $P_f(E)$ l'ensemble des parties finies de E.

Soient A et B deux ensemble, A/B dénote la soustraction ensembliste, $A \subseteq_f B$ signifie $A \in P_f(B)$.

Une *famille* $\{a_i\}_{i \in I}$ est un multi-ensemble d'objets a_i indexés sur un ensemble (dénombrable) d'indices I.
Un *vecteur* $\vec{a}$ de longueur n (nous noterons $long(\vec{a}) = n$) est une famille finie $\vec{a} = \{a_i\}_{i \in 1..n}$. Nous noterons souvent $\vec{a} = a_1, ..., a_n$.
Un vecteur $\vec{a} = \{a_i\}_{i \in 1..n}$ est dit *linéaire* si ses éléments sont deux à deux différents. C'est-à-dire si $\{a_i\}_{i \in 1..n}$ est un ensemble.
Une *structure algébrique* est un n-uplet $\langle E, f_1, ..., f_{n-1} \rangle$ où E est un ensemble non vide et les f_i sont des opérateurs ou des lois de composition interne sur E.

Soient E un ensemble, $*$ une loi de composition interne sur E et e un élément de E.
La structure $\langle E, *, e \rangle$ est un *monoïde* si et seulement si

- ▷ $\forall x \in E,\ e * x = x * e = x$
- ▷ $\forall x, y, z \in E,\ (x * y) * z = x * (y * z)$

$\langle E, *, e \rangle$ est un *monoïde commutatif* si et seulement si $\langle E, *, e \rangle$ est un monoïde et

- ▷ $\forall x, y \in E,\ x * y = y * x$

Considérons un ensemble E muni d'un ordre partiel $\leq$ (nous le noterons souvent $\langle E, \leq \rangle$) [2].
Soient $A \subseteq E$ et $x \in E$, x est un *majorant* de A si $\forall y \in A,\ y \leq x$. Un majorant x de A est *le plus petit majorant* (ppm) de A si pour tout majorant y de A on a $x \leq y$.
Soient $A \subseteq E$ et $x \in E$, x est un *minorant* de A si $\forall y \in A,\ x \leq y$. Un minorant x de A est *le plus grand minorant* (pgm) de A si pour tout minorant y de A on a $y \leq x$.
$A \subseteq E$ est une partie *dirigé* de E si toute partie finie de A possède un majorant dans A.
Un *treillis* est un ensemble muni d'un ordre partiel $\langle E, \leq \rangle$ qui vérifie : toute partie finie de E admet un plus petit majorant et un plus grand minorant. Un *treillis complet* est un ensemble muni d'un ordre partiel $\langle E, \leq \rangle$ qui vérifie : toute partie de E admet un plus petit majorant et un plus grand minorant.
Soit $\langle E, \leq \rangle$ un treillis complet, notons $\sqcup$ (resp. $\sqcap$) l'opérateur qui a une partie de E fait correspondre son plus petit majorant (resp. plus grand minorant), notons $\top$ (resp. $\perp$) le plus grand (resp. plus petit) élément de E. $\langle E, \leq \rangle$ peut être noté sous la forme de la structure algébrique $\langle E, \leq, \sqcup, \sqcap, \top, \perp \rangle$.

[2] E est un poset (partial order set)

Soit $\langle E, \leq, \sqcup, \sqcap, \top, \bot\rangle$ un treillis,
$x \in E$ est un élément *fini* de $\langle E, \leq, \sqcup, \sqcap, \top, \bot\rangle$ si pour toute partie dirigée D de E, on a : $x \leq \sqcup D \Rightarrow \exists y \in D$ *tel que* $x \leq y$,
$x \in E$ est un élément *compact* de $\langle E, \leq, \sqcup, \sqcap, \top, \bot\rangle$ si pour toute partie A de E, on a : $x \leq \sqcup A \Rightarrow \exists B \in P_f(A)$ *tel que* $x \leq \sqcup B$.
Dans les treillis complets, les éléments finis et les éléments compacts sont les mêmes.
Un treillis complet $\langle E, \leq, \sqcup, \sqcap, \top, \bot\rangle$ est dit *algébrique* si
$\forall x \in E,\, x = \sqcup\{y | y\ est\ fini\ et\ y \leq x\}$.
Un treillis complet est dit *ω-algébrique* s'il est algébrique et si l'ensemble de ses éléments finis est dénombrable.

2.2 Les systèmes de contraintes cylindriques avec éléments diagonaux

Nous proposons une formalisation des systèmes de contraintes qui est abstraite de toute considération constructive et se concentre sur la structure algébrique finale : un système de contraintes est un treillis complet ω-algébrique muni de structures supplémentaires (éléments diagonaux et opérateurs de cylindrification). Nous suivons en cela l'approche, en particulier, de [BPP96]. Aucune supposition concernant un langage de description des contraintes n'est effectuée. Afin de modéliser les variables locales d'un langage de programmation, nous avons besoin d'opérateurs permettant de “cacher” dans une contrainte l'information concernant une variable particulière. Nous introduisons dans ce but des opérateurs de cylindrification qui sont adaptés de la théorie des algèbres cylindriques ([HMT71]). Il en est de même des éléments diagonaux qui nous permettront de modéliser le passage de paramètres. Leur signification intuitive est l'égalité entre variables.

Définition 2.1 Un *système de contraintes cylindrique avec éléments diagonaux* est une structure $\langle C, \leq, V, \{\exists_x : C \to C\}_{x\in V}, \{d_{xy}\}_{x,y\in V}\rangle$ vérifiant :

- $\langle C, \leq\rangle$ est un treillis complet ω-algébrique. Nous notons $vrai$ son plus petit élément, $faux$ son plus grand élément, $\sqcup$ (resp. $\sqcap$) la loi de composition interne *plus petit majorant* (resp. *plus grand minorant*).

- V est un ensemble dénombrable de variables.

- $\{d_{xy}\}_{x,y \in V} \subseteq C$
- $\forall x, y, z \in V,\ \forall c, d \in C,$

    1. $\exists_x c \leq c$
    2. si $c \leq d$ alors $\exists_x c \leq \exists_x d$
    3. $\exists_x(\exists_y c) = \exists_y(\exists_x c)$
    4. $\exists_x(c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$
    5. $d_{xx} = vrai$
    6. si $z \neq x, y$ alors $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$
    7. si $x \neq y$ alors $c \leq d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$

□

Les éléments de C sont appelés *contraintes*.
Soit $\mathcal{C} = \langle C, \leq, V,\ \{\exists_x : C \to C\}_{x \in V},\ \{d_{xy}\}_{x,y \in V}\rangle$ un système de contraintes, $c \in \mathcal{C}$ signifie que c est une contrainte de $\mathcal{C}$.

Nous écrirons souvent "système de contraintes" pour "système de contraintes cylindrique avec éléments diagonaux".

Nous imposons la condition d'algébricité dans la définition des systèmes de contraintes par nécessité de constructivisme. Dans le contexte informatique qui est le nôtre, tout élément du système de contraintes doit être calculable, c'est à dire être le plus petit majorant d'une suite d'éléments finis du treillis.
De même, les contraintes que nous utiliserons pour programmer dans un langage de programmation avec contraintes sont les contraintes finies.

Les éléments diagonaux modélisent l'égalité entre variables. Les opérateurs de cylindrification modélisent la quantification existentielle.
Notons que les précédents axiomes des opérateurs de cylindrification sont vérifiés par la quantification existentielle du calcul des prédicats mais que toutes les propriétés de cette dernière ne sont pas conséquence de ces axiomes. Il en est de même en ce qui concerne les éléments diagonaux par rapport à l'égalité dans son interprétation standard. Comme nous l'avons déjà indiqué, cette formalisation des systèmes de contraintes est plus abstraite que celle de [JL87] – cela est d'ailleurs vrai pour la plupart des autres formalisations ([DGP92], par exemple). Elle

nécessite la supposition de propriétés moins fortes du système de contraintes, précisement celles dont nous aurons besoin pour décrire la sémantique des CC.

Plutôt que de parler, par analogie avec la terminologie liée aux langages du premier ordre, de variables libres et de variables liées, nous préférons définir les notions de *variables contraintes* et de *variables non contraintes*. En effet, dans un langage du premier ordre, une variable liée est une variable qui apparaît quantifiée dans une formule. Nous nous intéresserons ici aux variables qui n'ont aucune "incidence" sur une contrainte. Supposons que c soit une formule dans un langage du premier ordre, la signification analogue de variable *non contrainte* (dans c) serait : variable qui est liée ou n'apparaît pas dans c.

Définition 2.2 Soit $c \in \mathcal{C}$.
L'ensemble des *variables non contraintes* de c est $vnc(c) = \{x \in V \mid \exists_x c = c\}$.
l'ensemble des *variables contraintes* de c est $vc(c) = V/vnc(c)$. □

Les définitions suivantes nous permettront de rendre compte de la notion de renommage. Notons que la notion de substitution définie ci-dessous (définition 2.4) est en quelque sorte la notion usuelle de substitution restreinte au remplacement de variables par des variables.

Définition 2.3 Pour tout couple de variables (x, y) nous définissons un *opérateur de substitution* $s_x^y : C \to C$

$$s_x^y c \stackrel{def}{=} \begin{cases} c & si \;\; x = y \\ \exists_x(d_{xy} \sqcup c) & si \;\; x \neq y \end{cases}$$

Sa signification intuitive est de remplacer les occurences libres de x par y. □

Remarquons que $s_x^y c = c$ si $x \notin vc(c)$.

Malgré l'évidence de son énoncé, la preuve de la proposition suivante est non triviale.

Proposition 2.1 Soit $c \in \mathcal{C}$, soient $x, y, z \in V$ tels que $y \notin vc(c)$.
$s_y^z(s_x^y c) = s_x^z c$. □

Lemme 2.1 Soient $x, y \in V$,$d_{xy} = d_{yx}$ □

preuve :

Si $x = y$: trivial.

$$\begin{array}{llr} \text{Sinon, } d_{yx} & \leq d_{xy} \sqcup \exists_x(d_{yx} \sqcup d_{xy}) & \text{(ax. 7)} \\ & \leq d_{xy} \sqcup d_{yy} & \text{(ax. 6)} \\ & \leq d_{xy} & \text{(ax. 5)} \end{array}$$

Nous avons mutatis mutandis $d_{xy} \leq d_{yx}$

Donc $d_{xy} = d_{yx}$

$\square$

Lemme 2.2 Soient $x, y \in V$, $\exists_x d_{xy} = vrai$ $\square$

preuve :

1^{er} cas : $x = y$. Trivial.

$2^{ème}$: cas $x \neq y$

$$\begin{array}{lll} \exists_x d_{xy} & = \exists_x(d_{xy} \sqcup d_{xy}) & \\ & = \exists_x(d_{xy} \sqcup d_{yx}) & \text{(lemme 2.1)} \\ & = d_{yy} & \text{(ax. 6)} \\ & = vrai & \text{(ax. 5)} \end{array}$$

$\square$

preuve de la proposition 2.1 :

Nous sommes obligé de distinguer deux cas à cause des conditions d'application de l'axiome (6).

1. cas $y = x$ ou $z = y$: trivial.
2. cas $x = z$ (et $x \neq y$) :

Nous avons (par définition) $s_y^x(s_x^y c) = \exists_y(d_{xy} \sqcup \exists_x(d_{xy} \sqcup c))$ (i)

Par l'axiome (7), on a $c \leq d_{xy} \sqcup \exists_x(d_{xy} \sqcup c)$

donc, par ax. (2), $\exists_y c \leq \exists_y(d_{xy} \sqcup \exists_x(d_{xy} \sqcup c))$

Comme de plus $c = \exists_y c$, on obtient :

$c \leq s_y^x(s_x^y c)$.

D'autre part,

$$\begin{array}{lll} (i) & \Rightarrow s_y^x(s_x^y c) \leq \exists_y(d_{xy} \sqcup d_{xy} \sqcup c) & \text{(ax. 1)} \\ & \Rightarrow s_y^x(s_x^y c) \leq \exists_y(d_{xy} \sqcup c) & \\ & \Rightarrow s_y^x(s_x^y c) \leq \exists_y(d_{xy}) \sqcup c & \text{(car } \exists_y c = c \text{ et par ax. (4))} \\ & \Rightarrow s_y^x(s_x^y c) \leq c & \text{(lemme 2.2).} \end{array}$$

Finalement $s_y^x(s_x^y c) = c$

3. cas $x \neq z$, $x \neq y$ et $y \neq z$:

$$
\begin{array}{lll}
s_y^z(s_x^y c) & = \exists_y(d_{yz} \sqcup \exists_x(d_{xy} \sqcup c)) & \\
& = \exists_y(\exists_x d_{yz} \sqcup \exists_x(d_{xy} \sqcup c)) & \text{(lemme 2.2)} \\
& = \exists_y \exists_x(d_{yz} \sqcup d_{xy} \sqcup c) & \text{(ax. 4 et lemme 2.2)} \\
& = \exists_x \exists_y(d_{yz} \sqcup d_{xy} \sqcup c) & \text{(ax. 3)} \\
& = \exists_x(\exists_y(d_{yz} \sqcup d_{xy}) \sqcup c)) & \text{(ax. 4) et } y \notin vc(c) \\
& = \exists_x(d_{xz} \sqcup c)) & \text{(ax. 6) et } x, y \neq z \\
& = s_x^z c &
\end{array}
$$

□

Définition 2.4 Une fonction $\sigma : V \rightarrow V$ est une *subsitution* si elle est l'identité partout sauf sur un sous-ensemble fini de V.
Une substitution peut être représentée sous la forme $[y_1/x_1, ..., y_n/x_n]$.
Soit $\sigma = [y_1/x_1, ..., y_n/x_n]$ une subsitution.
L'ensemble $\{x_1, ..., x_n\}$ est appelé *domaine* de σ.
L'ensemble $\{y_1, ..., y_n\}$ est appelé *codomaine* de σ.
□

La notion de substitution est très générale. En particulier il n'est pas demandé qu'une substitution soit idempotente. Nous utiliserons au chapitre 5 des substitutions quelconques. Cela pose cependant des problèmes quant à leur application à des contraintes.
Considérons la contrainte $x = y$ (c'est-à-dire l'élément diagonal d_{xy}) et la substitution $[x/y, y/x]$. En supposant que l'application d'une substitution soit définie comme étant l'application de la composition des opérateurs de substitution correspondants, nous avons :

$$
\begin{array}{lll}
s_x^y(s_y^x d_{xy}) & = \exists_y(d_{yx} \sqcup \exists_x(d_{xy} \sqcup d_{xy})) & \\
& = vrai & \text{(par lemme 2.2)}
\end{array}
$$

Ce qui n'est bien évidemment pas le résultat espéré.

Ce problème provient du fait que l'application de la substitution a été effectué variable par variable en substituant successivement y par x puis x par y. Considérant une substitution portant sur plusieurs variables, les opérateurs de substitution risquent d'interférer les uns avec les autres lors de la substitution d'une variable contrainte par une autre variable contrainte, aboutissant ainsi à un résultat non souhaité. Une solution à ce problème consiste à utiliser un vecteur intermédiaire : les variables sont d'abord renommées avec des variables "fraîches" (n'apparaissant ni dans la contrainte ni dans la substitution). La substitution est alors appliquée à la contrainte ainsi "renommée", tout risque d'interférence parasite entre variables de même nom étant ainsi évité. Il est évident que le choix du vecteur intermédiaire est sans importance.

Définition 2.5 Etant donné une substitution $\sigma = [y_1/x_1, ..., y_n/x_n]$ et une contrainte c, nous noterons $c\sigma$ *l'application* de c à la composition d'opérateurs de substitution $s_{z_1}^{y_1} \circ ... \circ s_{z_n}^{y_n} \circ s_{x_1}^{z_1} \circ ... \circ s_{x_n}^{z_n}$, $\vec{z} = z_1, ..., z_n$ étant un vecteur linéaire de variables de longueur n tel que $\vec{z} \cap (\vec{y} \cup vc(c)) = \emptyset$.
C'est-à-dire $c\sigma = s_{z_1}^{y_1} \circ ... \circ s_{z_n}^{y_n} \circ s_{x_1}^{z_1} \circ ... \circ s_{x_n}^{z_n} c$. $\square$

Remarquons que si $\vec{y}$ vérifie $\vec{y} \cap vc(c) = \emptyset$ nous avons $c[\vec{y}/\vec{x}] = s_{x_1}^{y_1} \circ ... \circ s_{x_n}^{y_n} c$ (la preuve découle de la proposition 2.1).
Nous utiliserons par la suite cette propriété pour simplifier certaines preuves (en fait, nous serons toujours dans ce cas sauf au chapitre 5 lorsque nous manipulerons des *abstractions de contraintes*).

Remarquons aussi que cette notion de substitution restreinte au cas idempotent correspond à la notion standard de renommage. En particulier, tout risque de capture d'une variable contrainte par un opérateur de cylindrification est écarté.

La proposition suivante montre que le renommage s'effectue de façon cohérente par rapport aux opérateurs de cylindrification. Nous nous limitons au cas du renommage d'un seule variable.

Proposition 2.2 Soit $c \in C$, soit $\sigma = [y/x]$ une substitution vérifiant $y \notin vc(c)$,
$\exists_y c[y/x] = \exists_x c$ $\square$

preuve :
1^{er} cas : $x \notin cv(c)$ ou $x = y$
Nous avons $c[y/x] = c$ donc $\exists_y c[y/x] = \exists_y c$
Si $x = y$ on conclue directement.
Si $x \neq y$ et $x \notin cv(c)$ alors $\exists_y c = c$ et $\exists_x c = c$.
D'où $\exists_y c = \exists_x c$
$2^{ème}$ cas : $x \neq y$ et $x \in cv(c)$
Montrons que $\exists_x c = \exists_y \exists_x (d_{xy} \sqcup c)$.

$$
\begin{array}{lll}
\text{puisque } d_{xx} = vrai, \exists_x c & = \exists_x(d_{xx} \sqcup c) & \text{(ax. 5)} \\
& = \exists_x(\exists_y(d_{xy} \sqcup d_{yx}) \sqcup c) & \text{(ax. 6)} \\
& = \exists_x(\exists_y(d_{xy} \sqcup d_{xy}) \sqcup c) & \text{(lemme 2.1)} \\
& = \exists_x(\exists_y d_{xy} \sqcup c) & \\
& = \exists_x(\exists_y d_{xy} \sqcup \exists_y c) & y \notin cv(c) \\
& = \exists_x \exists_y (d_{xy} \sqcup \exists_y c) & \text{(ax. 4)} \\
& = \exists_x \exists_y (d_{xy} \sqcup c) & y \notin cv(c) \\
& = \exists_y \exists_x (d_{xy} \sqcup c) & \text{(ax. 3)}
\end{array}
$$

□

Remarquons aussi que les opérateurs de cylindrification sont idempotents.

Proposition 2.3 Soit $c \in C$, $\exists_x \exists_x c = \exists_x c$ □

preuve :

$$
\begin{array}{lll}
\exists_x \exists_x c & = \exists_x(vrai \sqcup \exists_x c) & \\
& = \exists_x vrai \sqcup \exists_x c & \text{(ax. 4)} \\
& = \exists_x c &
\end{array}
$$

□

2.2.1 Exemple : arbres finis (système de contraintes de Herbrand)

Le système de contraintes de Herbrand tient une place centrale dans la PLC et les CC. Il permet de généraliser le mécanisme d'unification de la PL et cette dernière peut être vue comme étant la PLC sur le système de contraintes de Herbrand.

Considérons un ensemble dénombrable V de variables, un ensemble dénombrable Π de symboles de fonctions auquel est associé une fonction d'arité, et le symbole de prédicat $=$. Notons $Term$ l'ensemble (dénombrable) des termes formés sur cette signature et *Atom* l'ensemble des atomes. Ces derniers sont des égalités entre termes.

A un ensemble d'atomes on fait correspondre un unificateur le plus général (qui est une substitution idempotente). Celui-ci est unique à un renommage près. Nous notons $\emptyset$ la substitution identité. Considérons la relation *est plus général que*, notée $\leq$, sur les substitutions ($\sigma \leq \theta$ ssi $\exists \rho$ tel que $\sigma\rho = \theta$) et la relation d'équivalence $\sim$ qui en est induite. Notons $Isubs|_\sim$ le quotient de l'ensemble des substitutions idempotentes par $\sim$.

Etant donné un ensemble d'atomes (ensemble d'équations), il est bien connu que tous les unificateurs les plus généraux sont équivalents par rapport à $\sim$.

Soit $\top$ un élément arbitraire qui n'appartient pas à $Isubs$. Notons $Isubs^\top$ l'ensemble $Isubs \cup \{\top\}$. Nous étendons $\leq$ et $\sim$ de façon à ce que $\top$ soit le plus grand élément de $Isubs^\top$. Notons $Isubs^\top|_\sim$ le nouvel ensemble quotient et, si σ est une substitution idempotente, $\sigma^\sim$ la classe d'équivalence de σ.

Proposition 2.4 (Eder [Ede85]) $\langle Isubs^\top|_\sim, \leq, \rangle$ est un treillis complet. son plus petit élément est $\emptyset^\sim$ et son plus grand élément $\top^\sim$. $\square$

Pour toute variable x, considérons l'opérateur $\exists_x : Isubs^\top \to Isubs^\top$ définit comme suit.
$\exists_x \top = \top$
Soit $\sigma \in Isubs$.
si x n'appartient pas au domaine de σ, alors $\exists_x \sigma = \sigma$.
si $\sigma = [..., x/t, ...]$, alors $\exists_x \sigma$ est obtenu à partir de σ en supprimant x/t et en remplaçant les occurences de x par t (t ne contient pas x car σ est idempotente).

Nous avons clairement, pour $\sigma, \sigma' \in Isubs^\top$ $\sigma \sim \sigma' \Rightarrow \exists\sigma \sim \exists\sigma'$.

Pour toute variable x, $\exists_x$ s'étend à $Isubs^\top|_\sim \to Isubs^\top|_\sim$ de manière évidente.

Proposition 2.5 La structure

$$\langle Isubs^\top|_\sim, \leq, V, \{\exists_x : Isubs^\top|_\sim \to Isubs^\top|_\sim\}_{x\in V}, \{[x/y]\}_{x,y\in V}\rangle$$

est un système de contraintes. $\square$

(schéma de) preuve :
Par proposition 2.4, $\langle Isubs^\top|_\sim, \leq, \rangle$ est un treillis complet.
L'algébricité découle du théorème 3.10 de [Ede85].
L'ensemble des éléments finis est dénombrable puisque $Term$ est dénombrable.
Nous avons donc bien un treillis complet ω-algébrique.
Il n'est pas difficile de vérifier que les axiomes de la définition 2.1 sont satisfaits.
$\square$

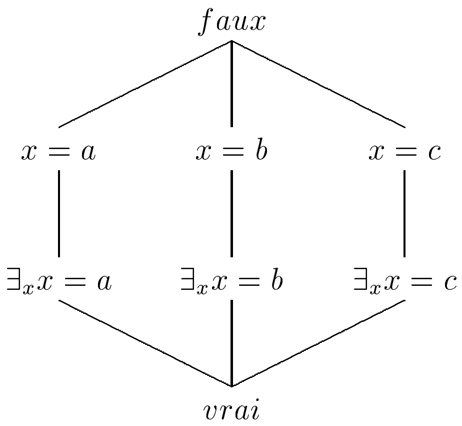

Figure 2.1 : Système de contraintes de Herbrand

La figure 2.1 montre le système de contraintes ainsi construit sur $V = \{x\}$ et $\Pi = \{a, b, c\}$, a, b et c étant des symboles de fonction d'arité 0 (i.e. ce sont des constantes). Le lecteur pourra aussi se reporter à la figure 2.2 qui montre le système de contraintes de Herbrand sur $V = \{x, y\}$ et $\Pi = \{a, b\}$ mais sans les opérateurs de cylindrification.

Remarquons que l'opérateur de cylindrification définit ci-dessus est plus faible que la quantification existentielle de la logique. Si $\sigma(A)$ est l'élément de $Isubs^{\top}|_{\sim}$ associé à un ensemble d'atomes A, nous avons par exemple $\exists_x \sigma(x = a) = \emptyset$. Cela est suffisant pour modéliser le fonctionnement du calcul en PL et en PLC sur le système de contraintes de Herbrand. En effet, l'utilisation (implicite en PL et PLC, explicite dans les CC) effectuée de cet opérateur dans les langages de programmation en logique ne correspond pas au concept déclaratif de quantification existentielle mais à la déclaration de variables locales. On pourrait remarquer aussi que l'opérateur plus grand minorant ne correspond pas au *ou* logique. Il serait bien sûr possible de définir un système de contraintes sur les égalités entre termes dont la relation de conséquence reflète la relation de conséquence logique et dont les opérateurs de cylindrification reflétent la quantification existentielle.

2.2.2 Extension distributive

Un tel système de contraintes n'est pas distributif en général, ce qui pose problème quant à la définition de la sémantique dénotationnelle des CC. De plus, les plus petit majorant et plus grand minorant ne correspondent pas aux conjonction et disjonction de la logique classique. Cela est mis en évidence par l'exemple suivant.

Considérons le système de contraintes représenté figure 2.1 dont l'ensemble de variables V est le singleton $\{x\}$. Le seul élément diagonal est d_{xx} qui est égal à $vrai$. L'opérateur de cylindrification $\exists_x$ étant idempotant, il n'y a que six contraintes en dehors de $vrai$ et $faux$.

Nous avons $(x = a \sqcap x = b) = vrai$, ce qui ne correspond pas à la disjonction de la logique classique.
D'autre part, $((x = a \sqcap x = b) \sqcup x = c) = (x = c)$ et $((x = a \sqcup x = c) \sqcap (x = b \sqcup x = c)) = faux$. Le plus petit majorant n'est donc pas distributif par rapport au plus grand minorant.
Du point de vue d'une sémantique dénotationnelle, cela pose un problème pour prendre en compte le non déterminisme (dont know) propre aux langages de programmation en logique et à leurs extentions. Par exemple, un langage de PLC sur le système de contraintes de l'exemple précédent pourra, à partir du store $x = c$, développer deux branches ajoutant les containtes $x = a$ et $x = b$ respectivement. Il est bien évident que dans ce cas, le résultat intuitif attendu est que les deux branches en question échouent. On aimerait donc que $(x = a \sqcap x = b) \sqcup x = c$ soit égal à $faux$.

Il est possible de répondre à ce problème en intégrant, en passant à l'ensemble des parties, les systèmes de contraintes dans une sur-structure distributive dans laquelle le ppm et le pgm modélisent la conjonction et la disjonction logique. N'en ayant pas l'utilité dans cette thèse, nous ne développons pas ici ce sujet. Le lecteur intéressé peut se reporter à [BPP96].

2.3 Approche constructive

Comme nous l'avons discuté dans le premier chapitre, la formalisation des systèmes de contraintes en terme de domaines de Scott ([Sco82]) permet de voir les CC (et la PLC) comme un paradigme de calcul par informations partielles. Cette approche est très attractive. D'une part, la notion de système d'information est un élément important de la théorie des langages de programmation. D'autre part, celle-ci nous permet un vision "constructive" des systèmes de contraintes. Nous avons formalisés ceux-ci dans la section précédente comme étant une structure qui vérifie, entre autres, la propriété d'être un treillis complet ω-algébrique. La construction d'une telle structure pour un domaine de contraintes particulier est souvent peu évidente. La construction de ce que nous avons appelé système de contraintes (cylindrique avec éléments diagonaux) peut s'effectuer par l'intermédiaire des systèmes d'information. Elle est alors basée sur un ensemble de *jetons*[3] ou *contraintes élémentaires* et une relation de conséquence qui vérifie certaines propriétés (en particulier, elle est compacte). Les contraintes sont les ensembles de jetons clos par la relation de conséquence. La relation de conséquence s'étend sur les contraintes et, l'ensemble des contraintes muni de cette relation forme un treillis complet ω-algébrique. Les contraintes finies étant les éléments compacts du treillis, il est alors possible de décrire une contrainte finie (et ce sont elles qui seront utilisée dans un langage concret) par un ensemble générateur de jetons (cet ensemble étant fini, par compacité de la relation de conséquence). Un système de contraintes est un système d'information de Scott étendu d'une manière similaire – mais plus constructive – à celle que nous avons utilisé sur les treillis avec des éléments diagonaux et des opérateurs de cylindrification.

Les domaines considérés ne sont en fait pas exactement des domaines de Scott mais ces derniers auquels la condition de cohérence[4] a été retirée. En effet, Scott ayant proposé les systèmes d'information dans le but de donner un modèle au lambda-calcul – et de définir une structure de domaines adaptée aux besoins de la sémantique dénotationnelle en général –, il ne s'est pas intéressé aux éléments incohérents. Dans le cadre CC, ceux-ci vont acquérir une importance particulière, la notion d'échec (fort différente de celle de la PL et de la PLC, nous y reviendrons) se concrétisant par l'obtention d'un store incohérent. Il n'y a qu'une contrainte (dite) incohérente (elle est en fait cohérente au sens des systèmes d'information) notée $faux$ qui est l'ensemble de tous les *jetons* (tout jeton est conséquence de $faux$).

[3]tokens

[4]Contrairement à l'usage oral, nous traduisons *consistence* par *cohérence*, évitant ainsi le néologisme de *consistance*. Il en va de même pour *consistent* que nous traduisons par *cohérent*.

2.3.1 Systèmes d'information

Notre présentation des systèmes d'information suit essentiellement [Sco82].

Définition 2.6 Un système d'information A est une structure
$A = \langle D, \Delta, Coh, \vdash \rangle$
dans laquelle :
D est un ensemble, $\Delta \in D$, $Coh \subseteq P_f(D)$, $\vdash \subseteq Coh \times D$ est une relation binaire.
Pour tout $u, v \in P_f(D)$, pour tout $X \in D$, Coh vérifie :

(i) si $v \in Coh$ et $u \subseteq v$ alors $u \in Coh$

(ii) si $X \in D$ alors $\{X\} \in Coh$

(iii) si $u \vdash X$ alors $u \cup \{X\} \in Coh$

Pour tout $u, v \in Coh$, pour tout $X \in D$, $\vdash$ vérifie :

(iv) $u \vdash \Delta$

(v) si $X \in u$ alors $u \vdash X$

(vi) si $u \vdash X$ et $\forall Y \in u\ v \vdash Y$ alors $v \vdash X$

□

Les éléments de D sont appelés *jetons*. Ils sont aussi appelés "objets informatiques" ou "propositions" dans [Sco82]. Ce sont des éléments basiques d'information. Un jeton – ou un ensemble de jetons – contient[5] de l'information concernant les *éléments* possibles du domaine à construire.

Coh est l'ensemble des parties finies cohérentes de D, c'est à dire celles dont les jetons sont "compatibles" entre eux. Cette notion reste ici très intuitive. Il faut bien sûr faire le rapprochement avec la notion de cohérence en logique. Cependant, Scott ne donne pas de critère de cohérence, il se contente de supposer l'existence de Coh qui est donc donné et non pas déduit de la structure. Scott ne s'est en fait pas intéressé aux parties incohérentes. Dans le cadre des CC, où la notion d'échec consiste en l'obtention d'une information incohérente,

[5]Nous avons choisi de traduire ici le verbe anglais "to involve" utilisé dans [Sco82] par "contenir".

nous ne nous intéresserons plus seulement aux parties cohérentes de D. C'est pourquoi nous définirons les systèmes de contraintes comme étant des systèmes d'information dans lesquels la référence à Coh est supprimé. Nous déduirons alors de la structure obtenue un ensemble d'*éléments cohérents*, l'idée étant que, par analogie avec la logique, les ensembles de jetons *incohérents* sont ceux dont tous les éléments de D sont conséquence.

A défaut d'une traduction satisfaisante de l'anglais "entailment relation", nous appelons $\vdash$ "relation de conséquence". Nous justifions ce choix par la signification intuitive de $u \vdash X$ qui est "l'ensemble de jetons u contient plus d'information que le jeton X", ou plutôt "u contient suffisament d'information pour déduire celle contenue par X". Si $u \vdash X$, nous dirons que "u à pour conséquence X" ou que "X est conséquence de u".

Les axiomes (v) et (vi) sont des formes de réflexivité et de transitivité respectivement. $\vdash$ peut s'étendre à $Coh \times Coh$ par $u \vdash v$ ssi $u \vdash X$ pour tout $X \in v$. Remarquons que cette extension est une relation de pré-ordre sur Coh.

Δ est conséquence de tout élément de Coh (axiome (iv)). Il est l'élément le moins informatif de D – celui qui contient le moins d'information. Scott écrit qu'il est l'élément qui contient *zéro information.*

Les éléments du système d'information sont les ensembles de jetons clos par la relation de conséquence et dont les parties finies sont cohérentes.

Définition 2.7 Les *éléments* d'un système d'information $A = \langle D, Coh, \Delta, \vdash \rangle$ sont les parties x de D vérifiant :

$$\forall u \in P_f(D), \text{ si } u \subseteq x \text{ alors } u \in Coh$$

$$\forall u \in P(D), \text{ si } u \subseteq x \text{ et } u \vdash X \text{ alors } X \in x$$

L'ensemble (noté $\mathbf{A}$) des éléments d'un système d'information A est un *domaine de Scott.* $\square$

Nous dirons qu'un ensemble de jetons (possiblement infini) est cohérent si toutes ses parties finies sont cohérentes – autrement dit, la propriété de cohérence est compacte.

Soit u une partie cohérente (possiblement infinie) de D. Nous notons $\uparrow u = \{X \in D | \exists v \subseteq_f u \textit{ tel que } v \vdash X\}$ la clôture de u par la relation de conséquence.

Par l'axiome (iii) et la définition 2.7 nous avons : $\uparrow u$ est un élément de A. Nous dirons que u *génère* l'élément $\uparrow u$.
Nous appelons *éléments finis* de A les élément de A générés par une partie finie (cohérente) de D.

Définition 2.8 Les éléments x de A vérifiant
$\exists u \subseteq Coh$ tel que $x = \uparrow u$
sont appelés *éléments finis* de A.
Nous notons $\mathbf{A_f}$ l'ensemble des éléments finis de A. □

La proposition suivante est due à Dana Scott ([Sco82]). Elle présentera un intérêt primordial dans le cadre de la définition des systèmes de contraintes à partir des systèmes d'information.

Proposition 2.6 (Scott, [Sco82])
Soit A un système d'information.
Si $\mathbf{A}$ possède un plus grand élément pour l'inclusion ensembliste alors $\langle \mathbf{A}, \subseteq \rangle$ est un treillis complet algébrique dont l'opérateur plus petit majorant est la clôture par la relation de conséquence de l'union ensembliste (i.e. $\sqcup = \uparrow \circ \cup$) et l'opérateur plus grand minorant est l'intersection ensembliste (i.e. $\sqcap = \cap$).
□

Le plus petit majorant n'est pas directement l'union ensembliste car l'union de deux éléments d'un domaine n'est pas forcément un élément de celui-ci (elle n'est pas forcément close par la relation de conséquence). Remarquons cependant que nous avons la propriété suivante ([Sco82]) :
pour tout élément x d'un domaine, $x = \cup\{\uparrow u \mid u \subseteq_f x \textit{ et } u \textit{ cohérent}\}$

Proposition 2.7 Soit $A = \langle D, \Delta, Coh, \vdash \rangle$ un système d'information.
Si $\mathbf{A}$ possède un plus grand élément pour l'inclusion ensembliste et si D est dénombrable alors $\langle \mathbf{A}, \subseteq \rangle$ est un treillis complet ω-algébrique. □

preuve :
Par proposition 2.6, $\langle \mathbf{A}, \subseteq \rangle$ est un treillis complet algébrique. Il reste à montrer que l'ensemble des éléments finis de $\langle \mathbf{A}, \subseteq \rangle$ est dénombrable.
C'est immédiat car si D est dénombrable, $P(D)$ l'est aussi.
□

Il y a correspondance entre les notions d'élément fini d'un système d'information et d'élément fini du treillis associé.

Proposition 2.8 Soit $A = \langle D, Coh, \Delta, \vdash\rangle$ un système d'information vérifiant :
$\mathbf{A}$ possède un plus grand élément pour l'inclusion ensembliste.
L'ensemble des éléments finis de $\langle \mathbf{A}, \subseteq\rangle$ est $\mathbf{A_f}$. □

preuve :
Les éléments finis d'un treillis complet sont ses éléments compacts.
1. Montrons que tout élément compact de $\langle \mathbf{A}, \subseteq\rangle$ est généré par une partie finie de D.
Soit x un élément compact de $\langle \mathbf{A}, \subseteq\rangle$.
Considérons l'ensemble $s = \{\{\uparrow X\} | X \in x\}$. Nous avons $s \in P(\mathbf{A})$ et $x = \sqcup s$ (car x est clos par la relation de conséquence). Puisque x est compact et $x \subseteq \sqcup s$, il existe $t \subseteq_f s$ tel que $x \subseteq \sqcup t$.
Montrons que $\sqcup t$ est un élément fini de A.
t est un ensemble fini d'éléments finis de A. Posons $t = \{y_i\}_{i=1..n}$. Pour tout $i \in 1..n$, considérons $u_i \in Coh$ tel que u_i génére y_i. Nous avons, par propriété de la clôture, $\sqcup t = \uparrow \cup\{\uparrow u_i\} = \uparrow \cup\{u_i\}$.
$\cup\{u_i\}$ est un ensemble fini. Donc $\sqcup t$ est un élément fini de A.
D'autre part, nous avons par construction $x = \sqcup s$ et $\sqcup t \subseteq \sqcup s$.
Donc $\sqcup t = \sqcup s$.
Et finalement, $x = \sqcup t$ et x est un élément fini de A.
2. Montrons que tout élément fini de A est un élément compact de $\langle \mathbf{A}, \subseteq\rangle$.

Soit $x = \uparrow \{X_i\}_{i=1..n}$ un élément fini de A. Soit $\{y_j\}_{j\in J}$ une partie (possiblement infinie) de $\mathbf{A}$ vérifiant $x \subseteq \sqcup\{y_j\}_{j\in J}$.
Pour tout $i \in 1..n$, par la compacité de $\vdash$, on a : il exists $J_i \subseteq_f J$ tel que $X_i \in \{y_j\}_{j\in J_i}$.
Considérons un ensemble $\{J_i\}_{i=1..n}$ dont les éléments vérifient la propriété précédente.
$\tilde{J} = \cup_{i=1}^{n} J_i$ est un ensemble fini d'indices et on a $x \subseteq \sqcup\{y_j\}_{j\in\tilde{J}}$
Donc x est un élément compact de $\langle \mathbf{A}, \subseteq\rangle$.
□

2.3.2 Systèmes de contraintes élémentaires

La définition ci-dessous des systèmes de contraintes élémentaires suit principalement [SRP91]. Comme nous l'avons déjà dit plus haut, cette définition est inspirée des systèmes d'information mais ne fait pas appel à un ensemble de parties finies cohérentes. Cette notion de système de contraintes élémentaire est en fait une restriction de celle de système d'information.

Définition 2.9 Un *système de contraintes élémentaire* est une structure
$\langle D, \vdash \rangle$
dans laquelle :
D est un ensemble (non vide, dénombrable) de *contraintes élémentaires.*
$\vdash \subseteq P_f(D) \times D$ est une *relation de conséquence* vérifiant pour tout $X \in D$ et pour tout $u, v \in P_f(D)$:

c_1 si $X \in u$ alors $u \vdash X$

c_2 si $u \vdash X$ et $\forall Y \in u\ v \vdash Y$ alors $v \vdash X$

□

Les *éléments* d'un système de contraintes élémentaire sont simplement le parties closes par la relation de conséquence.

Définition 2.10 Les *éléments* d'un système de contraintes élémentaire $\langle D, \vdash \rangle$ sont les parties c de D vérifiant :

$$\forall u \subseteq_f c, \text{ si } u \vdash X \text{ alors } X \in c$$

Nous notons $\mathbf{D}$ l'ensemble des éléments de $\langle D, \vdash \rangle$. □

Remarquons que D est lui-même un élément du système de contraintes $\langle D, \vdash \rangle$. Nous le notons $faux$.

Nous notons $vrai$ l'élément suivant : $vrai = \{X \in D \mid \emptyset \vdash X\}$.
$vrai$ est bien un élément, de par la transitivité de la relation de conséqence : supposons $u \subseteq_f vrai$ et $X \in D$ tels que $u \vdash X$, alors comme $\forall Y \in u$, $\emptyset \vdash Y$, on a par l'axiome (vi) $\emptyset \vdash X$.

La proposition suivante établit une correspondance entre l'ensemble des systèmes de contraintes élémentaires et l'ensemble des systèmes d'information vérifiant : $Coh = P_f(D)$.

Proposition 2.9 Soit $C = \langle D, \vdash \rangle$ un système de contraintes élémentaire. Soit Δ tel que $\Delta \notin D$. Notons D^Δ l'ensemble $D \cup \{\Delta\}$. Notons $\vdash^\Delta$ la plus petite extention de $\vdash$ à D^Δ vérifiant : $\forall u \in D^\Delta \; u \vdash \Delta$.
$A_C = \langle D^\Delta, \Delta, P_f(D^\Delta), \vdash^\Delta \rangle$ est un système d'information.
De plus, les éléments de C sont les éléments de A_C dans lesquels on supprime l'élément Δ. □

preuve :
Les axiomes $(i), (ii)$ et (iii) sont trivialement vérifiés. Δ vérifie (iv) par construction. (v) et (vi) sont conséquence de c_1 et c_2, et de la construction de l'extension de $\vdash$ à D^Δ.
Remarquons que tout élément de A_C contient Δ.
Soit x un élément de A_C, $x/\{\Delta\}$ est un élément de C (il est aisé de voir que $x/\{\Delta\}$ est clos par $\vdash$).
Réciproquement, si x est un élément de C, $x \cup \{\Delta\}$ est un élément de A_C (comme précédemment, par construction de $\vdash^\Delta$ à partir de $\vdash$).
□

Corollaire 2.1 Soit $C = \langle D, \vdash \rangle$ un système de contraintes élémentaire.
$\langle \mathbf{D}, \subseteq \rangle$ forme un treillis complet ω-algébrique. □

preuve :
Immédiat par les propositions 2.7 et 2.9.
□

2.3.3 Exemple : système de contraintes élémentaire de Herbrand

Nous donnons dans cette sous-section une version du système de contraintes de Herbrand en terme de système d'information.

Considérons un ensemble V de variables, un ensemble Π de symboles de fonctions auquel est associé une fonction d'arité, et le symbole de prédicat $=$. Notons $Term$ l'ensemble des termes formés sur cette signature et $Atom$ l'ensemble des atomes.

Le système de contraintes élémentaire de Herbrand est $\langle Atom, \vdash \rangle$,
$\vdash$ étant la plus petite relation vérifiant :

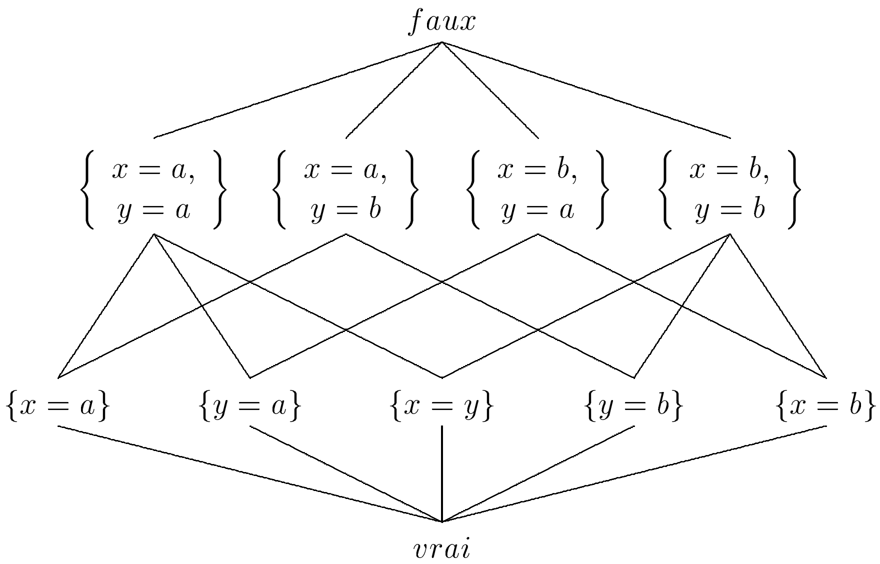

Figure 2.2 : Système de contraintes élémentaire de Herbrand

- les axiomes c_1 et c_2 de la définition 2.9.
- les axiomes de l'égalité de Clark [Cla79] :
  1. $\forall t \in Term.\ \emptyset \vdash \{t = t\}$
  2. $\forall s, t \in Term.\ \{s = t\} \vdash \{t = s\}$
  3. $\forall r, s, t \in Term.\ \{r = s, s = t\} \vdash \{r = t\}$
  4. si f est un symbole de fonction d'arité n alors
     $\{f(t_1, ..., t_n) = f(s_1, ..., s_n)\} \vdash \{t_1 = s_1, ..., t_n = s_n\}$
  5. si f est un symbole de fonction d'arité n alors
     $\forall i \in 1..n, \{t_i = s_i\} \vdash \{f(t_1, ..., t_n) = f(s_1, ..., s_n)\}$
  6. si f et g sont deux symboles de fonction distincts d'arité n et m respectivement alors
     $\forall E \in H,\ E \vdash \{f(t_1, ..., t_n) = g(s_1, ..., s_m)\}$
  7. si x apparait dans t et $t \neq x$ alors
     $\forall E \in H,\ E \vdash \{x = t\}$

La figure 2.2 montre le système de contraintes élémentaire de Herbrand avec $V = \{x, y\}$ et $\Pi = \{a, b\}$, a et b étant des symboles de fonction d'arité 0.

2.3.4 Systèmes de contraintes cylindriques avec éléments diagonaux

Il nous reste à définir les systèmes de contraintes proprements dits à partir des systèmes d'information. C'est-à-dire à introduire les opérateurs de cylindrification et les éléments diagonaux.

Définition 2.11 Un *pré-système de contraintes* (cylindrique avec éléments diagonaux) est une stucture $\langle D, \vdash, V, \{\exists_x : P_f(D) \rightarrow P_f(D)\}_{x \in V}, \{\delta_{xy}\}_{x,y \in V}\rangle$ vérifiant :

- $\langle D, \vdash \rangle$ est un système de contraintes élémentaire.
- V est un ensemble dénombrable de variables.
- $\{\delta_{xy}\}_{x,y \in V} \subseteq D$
- $\forall x, y, z \in V,\ \forall u, v \in P_f(D)$,
  1. $u \vdash \exists_x u$
  2. si $u \vdash v$ alors $\exists_x u \vdash \exists_x v$
  3. $\exists_x(\exists_y u) \approx \exists_y(\exists_x u)$
  4. $\exists_x(u \cup \exists_x v) \approx \exists_x u \cup \exists_x v$
  5. $\emptyset \vdash \delta_{xx}$
  6. si $z \neq x, y$ alors $\{\delta_{xy}\} \approx \exists_z\{\delta_{xz}, \delta_{zy}\}$
  7. si $x \neq y$ alors $\delta_{xy} \cup \exists_x(u \cup \delta_{xy}) \vdash u$

□

$u \approx v$ signifie $u \vdash v$ et $v \vdash u$.

Pour toute variable x, $\exists_x$ est étendu de manière évidente en une fonction continue de $\mathbf{D}$ dans $\mathbf{D}$:
$\exists_x c = \{X \in D \mid \textit{il existe } u \subseteq_f c \textit{ tel que } \exists_x u \vdash X\}$
Pour tout couple (x, y) de variables, nous notons d_{xy} l'élément de $\langle D, \vdash\rangle$ généré par δ_{xy}

Proposition 2.10 Soit $\langle D, \vdash, V, \{\exists_x : P_f(D) \rightarrow P_f(D)\}_{x\in V}, \{\delta_{xy}\}_{x,y\in V}\rangle$ un pré-système de contraintes.
$\langle \mathbf{D}, \subseteq, V, \{\exists_x : \mathbf{D} \rightarrow \mathbf{D}\}_{x\in V}, \{d_{xy}\}_{x,y\in V}\rangle$ est un système de contraintes. □

preuve :
$\langle \mathbf{D}, \subseteq\rangle$ est un treillis ω-algébrique (corolaire 2.1).
Les axiomes 1. à 7. de la def. 2.1 sont satisfaits. Nous détaillons la preuve pour l'axiomes 1.
Nous notons p1.,...,p7. les axiomes de la définition 2.11.
1. : Soient $c \in \mathbf{D}$ et $x \in V$. Soit $X \in \exists_x c$.
Soit $u \subseteq_f c$ tel que $\exists_x u \vdash X$. Nous avons $u \vdash X$ (ax. p1. et c_2).
Donc $X \in u$ et $\exists_x u \subseteq u$.
□

2.3.5 Construire les opérateurs de cylindrification et les éléments diagonaux

Les systèmes d'information nous permettent la construction d'un treillis qui sert de base à un système de contraintes. On aimerait cependant aller plus loin dans cette approche et définir une restriction des systèmes d'information permettant d'obtenir directement un système de contraintes cylindrique avec éléments diagonaux. Cependant les opérateurs de cylindrification se définissent sur les ensembles de jetons et non pas sur les jetons eux-mêmes. Il semble donc difficile de raffiner la définition 2.11 dans ce but. Nous proposons dans cette sous-section une approche différente de ce problème. Une structure de système de contraintes est construite à partir d'un treillis complet ω-algébrique. Ce dernier pouvant lui-même être construit par l'intermédiaire des systèmes d'information.

Soit $\langle C, \leq\rangle$ un treillis complet ω-algébrique. Soit V un ensemble (dénombrable) de variables.

Considérons l'ensemble C' vérifiant :
C' est le plus petit ensemble définit par :
$C' = C \cup \{\exists_x c \mid x \in V, c \in C'\} \cup \{d_{xy} \mid x, y \in V\}$,
et tel que les éléments finis de $\langle C, \leq \rangle$ satisfont aux axiomes 1. à 7. de la définition 2.1 et à l'axiome suivant :

8. Si $\{c_i\}_i$ est une suite croissante alors $\exists_x \sqcup_i c_i = \sqcup_i \exists_x c_i$

L'axiome 8. ci-dessus à été proposé dans [BPP95]. Il impose aux opérateurs de cylindrification d'être continus et permet d'assurer que la structure de treillis complet ω-algébrique est préservé par la construction de C'.

Soit $x \in V$. Notons $\exists_x$ l'opérateur qui à un élément c de C' fait correspondre $\exists_x c$. La structure $\langle C', \leq, V, \{\exists_x : C' \to C'\}_{x \in V}, \{d_{xy}\}_{x,y \in V}\rangle$ définie par la construction précédente est un système de contraintes.

L'intérêt de cette construction est essentiellement théorique. Cela ne permet pas en général, étant donné un système de contraintes quelconque, de consruire en pratique les opérateurs de cylindrification.

2.3.6 Autres formalisations des systèmes de contraintes

Nous classons les différentes approches de la formalisation des systèmes de contraintes en deux grandes familles que nous qualifions d'*approche logique* pour la première et d'*approche algébrique* pour la seconde.
Dans l'approche logique, un système de contraintes est un langage du premier ordre associé à une interprétation (ou un ensemble d'interprétations) ou une théorie du premier ordre. On la retrouve en particulier dans [JL87]. Signalons que [DGP92] propose une vision intéressante de la PLC en tant qu'extension logique d'un système de contraintes qui lui-même est définit par un ensemble de formules atomiques et un ensemble d'interprétations.
L'approche algébrique, qui est celle que nous avons suivi, est plus abstraite. La notion de système de contraintes y est directement définie comme étant une structure algébrique (ou par l'intermédiaire de systèmes d'information) tandis que cette dernière est déduite de la structure logique dans le cas de l'approche logique. Notons que Saraswat à proposé une formalisation des systèmes de contraintes ([Sar92a]) qui part d'un ensemble de formules atomiques clos par conjonction et quantification existentielle et d'une relation de conséquence qui

satisfait à un ensemble de règles d'inférences. Il montre que les systèmes de contraintes ainsi définis forment une catégorie.

Chapitre 3

Les langages Concurrents avec Contraintes

Les langages concurrents avec contraintes (CC) ont été proposés à la fin des années 80 comme une généralisation de la programmation logique avec contraintes (PLC) d'une part et des langages logiques concurrents (LLC) d'autre part. Ils permettent de combiner les avantages des deux approches, c'est-à-dire d'une part la possibilité de raisonner (symboliquement) et calculer (numériquement) sur des domaines spécifiques (PLC) et d'autre part la possibilité d'avoir un contrôle dynamique des calculs dirigé par les données (LLC) et non plus dirigé par les instructions du programme.

L'idée fondamentale des langages CC est l'utilisation de contraintes pour étendre les mécanismes de synchronisation et de contrôle des langages logiques concurrents. Ainsi différents agents (processus) communiquent et se synchronisent par l'intermédiaire d'un ensemble global de contraintes appelé *store* dans lequel l'information s'accumule de manière monotone au fil du temps. Chaque agent peut soit ajouter une nouvelle contrainte (opération *tell*), soit demander si une certaine contrainte est vraie dans (i.e. impliquée par) le store (opération *ask*). La synchronisation s'effectue par une opération *ask bloquante*: si l'on ne peut décider si la contrainte est vraie ou fausse dans le store (i.e. il faut attendre d'avoir plus d'information), l'agent est suspendu jusqu'à ce que d'autres agents ajoutent (*tell*) les contraintes suffisantes pour décider.

Bien que se situant dans la lignée des langages de programmation logique – dont il est issu –, ce paradigme de calcul a perdu dans son intitulé le mot *logique*. En effet, les programmes CC ne possèdent pas d'interprétation dans la logique classique à cause de l'opérateur *ask*. Celui-ci, qui était noté $ask(c) \to A$ dans le travail originel de Saraswat, ne correspond pas à l'implication logique. Remarquons d'ailleurs que cet agent se réduit en *vrai* dans la cas où le store n'accepte pas le contrainte c (i.e. $store \sqcup c = faux$). Il nous paraît cependant utile d'insister sur le fait que les CC généralisent la Programmation Logique, la CLP et les LLC.

Comme dans le cas de la CLP, un langage concurrent avec contraintes particulier est défini par rapport à un (ou un ensemble de) système(s) de contraintes – on ecrira souvent CC(X), X étant un (ensemble de) système de contraintes. Le vif intérêt porté sur le paradigme CC est dû à plusieurs facteurs. Les CC étendent d'une manière naturelle la PL mais aussi la CLP dont le succès – y compris dans le domaine industriel – est avéré. Ils ouvrent la voie à une nouvelle méthodologie de programmation. Leurs fondements théoriques sont simples et bien établis. En particulier, ils disposent d'une sémantique dénotationelle en terme d'opérateurs de clôture. Nous ne développerons cependant pas cette dernière dans le cadre de cette thèse.

On peut définir les CC avec plusieurs opérateurs de choix différents : l'opérateur de choix aveugle (blind choice), l'opérateur de choix à un pas (one step choice) et l'opérateur de choix gardé (guarded choice). Nous les notons respectivement $+_a$, $+_1$, Σ_g.
L'opérateur de choix aveugle choisit l'une de ses opérandes sans aucune condition concernant celles-ci. $A +_a B$ peut se réduire en A ou en B, quels que soient A et B.
L'opérateur de choix à un pas ne choisi une de ses opérandes que si celle-ci n'est pas échouée.
L'opérateur de choix gardé porte sur des agents *gardés* par un opérateur *ask*. Une opérande ne peut être choisie que si sa garde est satisfaite.
Les CC sont le plus souvent présentés avec l'opérateur de choix aveugle ou l'opérateur de choix gardé. L'opérateur de choix aveugle a l'intérêt de permettre la définition d'une sémantique dénotationnelle des CC angéliques (cf. [JSS91]). L'opérateur de choix gardé est particulièrement intéressant dans le cas des implantations indéterministes pour lesquels un "mauvais" choix peut entrainer un échec. Nous considérerons dans cette thèse l'opérateur de choix à un pas qui facilitera dans les chapitre suivants la comparaison des langages que nous définirons avec le π-calcul. L'introduction d'un opérateur de choix gardé dans les MCC et les DCC (chapitres 5 et 6) fera l'objet de travaux futurs.

3.1 Syntaxe

La syntaxe des CC est très générale. Alors que la Programmation Logique ainsi que ses extensions que sont la CLP et les LLC est le plus souvent restreinte aux clauses de Horn (ou clauses normales pour la PL avec négation), la syntaxe des CC accepte une grande variété de formules. Cela a, en particulier, des implications quand à la quantifications des variables.
Pour les clauses de Horn (et les clauses normales), les quantifications existentielles et universelles sont implicites : les variables présentes dans le corps de la clause et non dans la tête sont quantifiées existentiellement et les autres universellement. Dans les CC, il en est de même pour la quantification universelle. Cependant, ils permettent une utilisation quelconque de l'opérateur de restriction (en particulier l'imbrication avec des opérateurs de choix ou de composition parallèle), laissant ainsi une grande liberté dans la définition de variables locales. Possibilité est aussi laissée au programmeur d'utiliser des variables globales (variables non quantifiées dans l'agent initial ou variables non quantifiées présentes dans le corps d'une déclaration et pas dans sa tête). Nous pensons cependant que l'emploi de telles variables globales n'est pas souhaitable d'un point de vue méthodologie de programmation, et tout particulièrement dans un cadre de programmation modulaire, les risques de surcharge de noms de variables étant évidents.

Syntaxe abstraite :

$$
\begin{array}{lcll}
P & ::= & Dec.A & \text{[programme et agent initial]} \\
Dec & ::= & \epsilon \mid p(\vec{x}) :: A \mid Dec.Dec & \text{[déclarations]} \\
A & ::= & c \mid ask(c).A & \\
 & & \mid A \parallel A \mid A + A & \text{[agents]} \\
 & & \mid p(\vec{x}) \mid \exists_x A &
\end{array}
$$

c est une contrainte finie, x est une variable, $\vec{x}$ est un vecteur de variables, p est un symbole de procédure appartenant à un ensemble dénombrable de symboles de procédures auquel est associé une fonction d'arité. Cet ensemble est disjoint de l'ensemble des variables.

Un programme est constitué d'un ensemble de déclarations (de procédures) et d'un agent que nous appelons *agent initial* ou *but*, pour reprendre la terminologie de la programmation en logique.

Etant donné un programme $P = D.B$, pour chaque symbole de procédure p d'arité n apparaissant dans un agent de P, il existe dans D une unique déclaration $p(x_1, ..., x_n) :: A$.

c correspond à l'opérateur *tell* dans sa version "eventual". Il a pour signification intuitive d'ajouter la contrainte c au store sans effectuer de test de cohérence. La version "atomic" de cet opérateur ([Sar89]) effectue ce test. Nous ne la considérerons pas ici.
Nous faisons le choix syntaxique de noter directement c (et non pas $tell(c)$) l'opérateur *tell* sur la contrainte c.
$ask(c).A$ est l'opérateur *ask*. Il est "bloquant" dans le sens où il attend que le store contienne suffisament d'information pour déduire c. Dans ce cas, $ask(c).A$ se réduit en A.
$||$ est la *composition parallelle*.
$+$ est l'*opérateur de choix à un pas*. Cet opérateur étant associatif et commutatif (voir plus loin), nous noterons souvent $A_1 + ... + A_n$ par $\Sigma_1^n A_i$ qu'il convient de ne pas confondre avec les opérateurs de choix gardé présents dans certaines présentations des CC.
La composition parallèle et le choix correspondent aux conjonction et disjonction (plusieurs clauses pour définir un même prédicat) de la programmation logique.
$p(\vec{x})$ est un appel de procédure.
$\exists$ est appelé *opérateur de restriction*. Sa signification est de déclarer une variable locale. Il y a ici surcharge de notation avec l'opérateur de cylindrification des systèmes de contraintes. Cela porte quelquefois à confusion. Nous conservons cependant cette notation car elle est utilisée habituellement dans le cadre des langages concurrents avec contraintes. La portée de cet opérateur dans $\exists_x A$ est l'agent A. Les notions de variables *libres* et *liées* sont définies de manière standard, les variables contraintes d'un agent c (c étant une contrainte) étant alors qualifiés de "libres". Nous écrirons parfois $\exists_{x,y}$ au lieu de $\exists_x \exists_y$. Si $\vec{x} = x_1, ..., x_n$ alors $\exists_{\vec{x}}$ dénote $\exists_{x_1} ... \exists_{x_n}$.

Les agents de type $c, ask(c).A$ et $p(\vec{x})$ sont appelés *agents simples*.

Etant donné un système de contraintes $\mathcal{C}$ et un ensemble de symboles de procédures Π, nous notons $Agents^{\mathcal{C},\Pi}$ l'ensemble des agents définit par cette syntaxe. Nous noterons celui-ci $Agents$ lorsque cela ne génére pas d'ambiguïté.

3.2 Sémantique opérationnelle

Nous décrivons la sémantique opérationnelle des CC dans un style inspiré de la *Machine Abstraite Chimique (CHAM)* ([BB92]). Nous ne conservons en fait de la CHAM que l'idée de travailler modulo une congruence structurelle sur les agents qui prend en compte les aspects syntaxiques que sont par exemple la commutativité des opérateurs ou l'α-renommage. Les règles de réduction se situent au niveau des agents simples et des règles structurelles permettent de dériver les réductions d'agents complexes. Cela revient à définir un système de transitions par l'intermédiaire de règles d'inférence de type SOS (e.g. [Plo81]) sur l'ensemble quotient des agents par la congruence structurelle.

3.2.1 Préliminaires

Nous rappelons certaines notions de base concernant les sémantiques opérationnelles structurelles (SOS)[1] et plus particulièrement les systèmes de transitions. Pour une étude approfondie, le lecteur peut se reporter à [Plo81].

Systèmes de transitions

Définition 3.1 Un *système de transitions* est une structure $\langle \Gamma, \rightarrow \rangle$ dans laquelle Γ est un ensemble d'éléments γ appelés *états* et $\rightarrow \subseteq \Gamma \times \Gamma$ est une relation binaire appelée *relation de transition*. $\Box$

Par la suite, nous écrirons $\gamma \rightarrow \gamma'$ pour $(\gamma, \gamma') \in \rightarrow$. Nous dirons que $\gamma \rightarrow \gamma'$ est une *transition*. Nous noterons $\rightarrow^*$ la clôture transitive de $\rightarrow$.
$\gamma \not\rightarrow$ signifie qu'il n'existe pas d'état γ' tel que $\gamma \rightarrow \gamma'$.
$\gamma \xrightarrow{n} \gamma'$, pour $n \in \mathrm{N}$, signifie qu'il existe une suite de n transitions $\gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_n$ vérifiant $\gamma_0 = \gamma$ et $\gamma_n = \gamma'$.

Définition 3.2 Un état γ est dit *terminal* s'il n'existe pas d'état γ' tel que $\gamma \rightarrow \gamma'$ (i.e. $\gamma \not\rightarrow$). $\Box$

Définition 3.3 Soit γ un état. Une *dérivation pour* γ est une suite d'états $\gamma_0, \gamma_1, \ldots$ telle que $\gamma_0 = \gamma$ et pour tout i, $\gamma_i \rightarrow \gamma_{i+1}$.
Une dérivation est *finie* s'il s'agit d'une suite finie.
Une dérivation finie $\gamma_0, \gamma_1, \ldots, \gamma_n$ est *terminée* si γ_n est un état terminal. $\Box$

[1] Structural Operational Semantics

Une dérivation finie $\gamma_0, \gamma_1, ..., \gamma_n$ est dite *de longueur n*. Nous noterons parfois $\gamma_0 \to \gamma_1 \to ... \to \gamma_n$ la dérivation $\gamma_0, \gamma_1, ..., \gamma_n$.

Définition 3.4 Un *calcul* pour γ est une dérivation pour γ qui est soit terminée soit non finie. □

Les définitions suivantes introduisent les notions de *dérivatif* et d'*arbre de dérivation.*

Définition 3.5 Soit γ un état, γ' est un *dérivatif immédiat* de γ s'il existe une transition $\gamma \to \gamma'$.
γ' est un *dérivatif* de γ s'il existe une suite de transitions $\gamma \to^* \gamma'$.
□

Définition 3.6 Soit γ un état. Notons D_γ l'ensemble des dérivatifs de γ.
Soit $E_\gamma = D_\gamma \cup \{\gamma\}$.
L'*arbre de dérivation* pour γ est le graphe orienté dont l'ensemble des noeuds est E_γ et les arcs sont les éléments de la relation $\to$ réduite à E_γ. □

L'arbre de dérivation n'est véritablement un arbre que si le graphe est acyclique, ce qui sera le cas pour les systèmes de transitions que nous considérerons par la suite.
Dans ce cas, il apparaît clairement que les branches finies (resp. infinies) correspondent aux dérivations terminées (resp. non finies).

Propriétés des systèmes de transitions

Nous rappelons la notion de confluence et définissons celle de *confluence uniforme.*

Définition 3.7 Un système de transitions $\langle \Gamma, \to \rangle$ est dit *confluent* s'il vérifie pour tout $\gamma, \gamma_1, \gamma_2 \in \Gamma$:
Si $\gamma \to^* \gamma_1$ et $\gamma \to^* \gamma_2$ alors il existe $\delta \in \Gamma$ tel que $\gamma_1 \to^* \delta$ et $\gamma_2 \to^* \delta$. □

Définition 3.8 Un système de transitions $\langle \Gamma, \to \rangle$ est dit *uniformément confluent* s'il vérifie pour tout $\gamma, \gamma_1, \gamma_2 \in \Gamma$:
Si $\gamma \to \gamma_1$ et $\gamma \to \gamma_2$ alors il existe $\delta \in \Gamma$ tel que $\gamma_1 \to \delta$ et $\gamma_2 \to \delta$. □

La confluence uniforme est une restriction de la *confluence forte* définie par Huet dans [Hue80]. Un système de transitions $\langle \Gamma, \rightarrow \rangle$ est *fortement confluent* s'il vérifie pour tout $\gamma, \gamma_1, \gamma_2 \in \Gamma$:
Si $\gamma \rightarrow \gamma_1$ et $\gamma \rightarrow \gamma_2$ alors il existe $\delta \in \Gamma$ tel que $\gamma_1 \rightarrow^* \delta$ et $\gamma_2 \rightarrow^* \delta$.

D'autre part, cette notion est similaire à la confluence uniforme définie par Niehren dans [Nie95] et détaillée dans sa thèse ([Nie94]) qui est malheureusement rédigée en langue allemande. Nous lui empruntons l'appelation “confluence uniforme”.

Nous montrons qu'un système de transitions uniformément confluent est confluent. La preuve est inspirée de [Hue80] dans lequel il est montré que la confluence forte implique la confluence.

Lemme 3.1 ([Hue80])
Un système de transitions $\langle \Gamma, \rightarrow \rangle$ est *confluent* si et seulement si il vérifie pour tout $\gamma, \gamma_1, \gamma_2 \in \Gamma$:
Si $\gamma \rightarrow \gamma_1$ et $\gamma \rightarrow^* \gamma_2$ alors il existe $\delta \in \Gamma$ tel que $\gamma_1 \rightarrow^* \delta$ et $\gamma_2 \rightarrow^* \delta$. □

preuve :
Par induction sur n, avec $\gamma \xrightarrow{n} \gamma_2$ □

Proposition 3.1 Soit T un système de transitions.
Si T est uniformément confluent alors T est confluent. □

(Schéma de) preuve :
Soit $T = \langle \Gamma, \rightarrow \rangle$.
On montre par une induction (facile) sur n que :
$\forall \gamma, \gamma_1, \gamma_2 \in \Gamma$, $\forall n$, si $\gamma \rightarrow \gamma_1$ et $\gamma \xrightarrow{n} \gamma_2$ alors $\exists \delta$ tel que $\gamma_1 \xrightarrow{n} \delta$ et $\gamma_2 \rightarrow \delta$.
Le résultat est alors obtenu par le lemme 3.1. □

Un système de transitions uniformément confluent et acyclique vérifie la propriété – très intéressante quand il sagit de modéliser une notion de calcul – d'être uniforme en compléxité.

Proposition 3.2 Soit $T = \langle \Gamma, \rightarrow \rangle$ un système de transitions uniformément confluent et acyclique. Pour tout $\gamma \in \Gamma$, tous les calculs pour γ sont de même longueur.
En particulier, s'il existe une dérivation non finie pour γ alors touts les calculs pour γ sont non finis. □

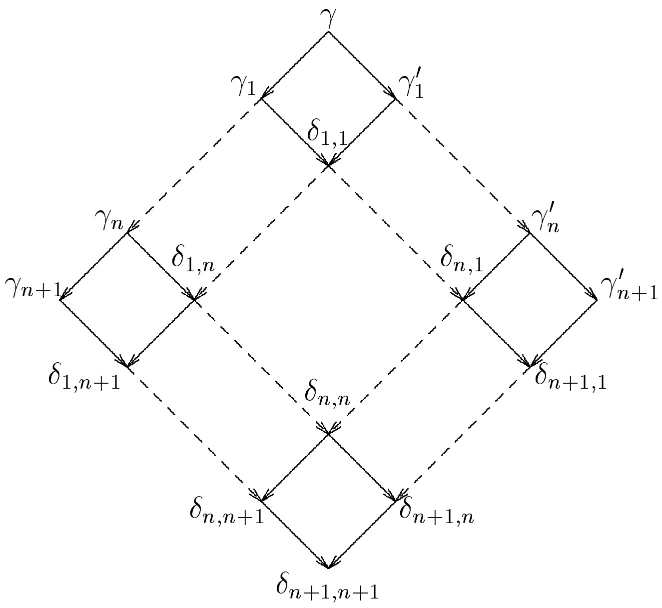

Figure 3.1 : Preuve de la prop. 3.2

preuve :
Soit $\gamma \rightarrow \gamma_1 \rightarrow ... \rightarrow \gamma_n$ un calcul de longueur n (on a donc $\gamma_n \not\rightarrow$).
Supposons qu'il existe une dérivation $\gamma \rightarrow \gamma'_1 \rightarrow ...$ (possiblement non finie) de longueur supérieure à n.
La figure 3.1 montre que puisque T est uniformément confluent, il existe une transition à partir de γ_n, ce qui contredit l'hypothèse $\gamma_n \not\rightarrow$.
En effet, par propriété de confluence uniforme, on construit les $\delta_{i,j}$ pour $i, j \leq n$.
Puis on construit $\delta_{n+1,1}$ jusqu'à $\delta_{n+1,n+1}$. Enfin, on construit $\delta_{n,n+1}$ jusqu'à γ_{n+1}.
$\square$

Définitions de type SOS

Une méthode standard pour spécifier un système de transitions est d'utiliser un système formel de règles d'inférences pour induire la structure sur les états. Cela se fait en général en basant l'induction sur la syntaxe abstraite du langage. Nous aurons par exemple dans le cas des CC la règle d'induction suivante :

$$\frac{P \rightarrow P'}{P \parallel Q \rightarrow P' \parallel Q} \quad \text{(parallel)}$$

Sa signification est que si la transition $P \rightarrow P'$ existe alors il est possible d'introduire l'opérateur de composition parallèle. Pour tout agent Q on a : $P \parallel Q \rightarrow P' \parallel Q$ est une transition.
Cette appproche permet une définition compacte de la sémantique opérationnelle. Elle permet aussi l'utilisation de méthodes de preuve simples et puissantes par induction structurelle.

3.2.2 Congruence structurelle

Nous définissons une relation de congruence sur les agents qui caractérise le fait que deux agents sont équivalents de par des aspects "syntaxiques" (e.g. commutativité des constructeurs $\parallel$ et $+$), par renommage des variables liées, ou par équivalence de contraintes.

Etant donné un agent A, et deux variables x et y. $A[x/y]$ dénote le renommage des occurrences libres de x par y dans A. Les contraintes apparaissant dans A sont renommées suivant le mécanisme décrit au chapitre 2.

La congruence structurelle $\equiv$ sur les agents est la plus petite relation de congruence vérifiant les axiomes suivants :

- $\langle Agents, \parallel, vrai \rangle$ et $\langle Agents, +, vrai \rangle$ sont des monoïdes commutatifs.
- $\exists_x A \equiv \exists_y A[y/x]$ si y n'est pas libre dans A
  $\exists_x A \parallel B \equiv \exists_x (A \parallel B)$ si x n'est pas libre dans B
  $\exists_x A + B \equiv \exists_x (A + B)$ si x n'est pas libre dans B
  $\exists_x \exists_y A \equiv \exists_y \exists_x A$
  $\exists_x A \equiv A$ si x n'est pas libre dans A
- $c \parallel d \equiv d \sqcup c$

Si $A \equiv B$ nous dirons que A et B sont structurellement équivalents.

L'existence pour tout agent de formes prenexes structurellement équivalentes est évidente. Celles-ci sont obtenues par renommage des variables liées avec de nouvelles variables et par extension de la portée de l'opérateur de restriction.

Définition 3.9 Un agent A est en *forme standard* s'il est de la forme :
$A = \exists_{x1} ... \exists_{xk}(d \parallel A_1 \parallel ... \parallel A_n)$
où d est une contrainte et les A_i sont :
soit des agents simples de type *ask* ou appel de procédure (i.e. ce ne sont pas des contraintes),
soit des agents ΣB_i. $\Box$

Proposition 3.3 Pour tout agent, il existe un agent en forme standard structurellement équivalent. $\Box$

preuve : par induction structurelle. $\Box$

Remarque : La forme standard n'est pas unique. En fait, pour tout agent, il existe une infinité d'agents en forme standard structurellement équivalents.

Remarquons aussi que l'équivalence structurelle est décidable. Une forme normale peut être obenue par un système finitaire de règles en supposant un ordre sur les variables et un ordre sur les agents (e.g. l'ordre lexicographique, en supposant que les agents sont écrits selon une certaine syntaxe sur un alphabet).

La notion de *store*, que l'on retrouve aux fondements mêmes des langages CC, correspond aux contraintes qui ont été ajoutées par le calcul (tell). Etant donné un état, son store associé est le plus petit majorant de ses contraintes en composition parallèle de premier niveau. Le store observable d'un état est la projection de son store sur ses variables globales.

Définition 3.10 Soit $A = \exists_{x1} ... \exists_{xk}(d \parallel A_1 \parallel ... \parallel A_n)$ un agent en forme standard.
Le *store* de A est : $store(A) \stackrel{def}{=} d$.
le *store observable* (ou *observable*) de A est: $obs(A) \stackrel{def}{=} \exists_{x1} ... \exists_{xk} d$. $\Box$

Remarque : dans la ligne précédente, $\exists$ est l'opérateur de cylindrification du système de contraintes, à ne pas confondre avec les opérateurs de restriction dans A. Cela ne génère pas d'ambiguité mais rend parfois la compréhension des agents difficile. Nous surchargons néanmoins le symbole $\exists$ car cette notation introduite par Saraswat est d'usage dans les langages CC.

Proposition 3.4 Deux agents structurellement équivalents en forme standard ont le même store observable. $\Box$

preuve : par induction structurelle. $\square$

La proposition précédente nous permet d'étendre la notion de store observable à l'ensemble des agents.

Définition 3.11 Soit A un agent qui n'est pas en forme standard, $obs(A) \stackrel{def}{=} obs(A')$, pour un agent A' en forme standard tel que $A \equiv A'$. $\square$

Proposition 3.5 Soient A et B deux Agents, nous avons les propriétés suivantes :

1. Si $A \equiv B$ alors $obs(A) = obs(B)$
2. $obs(A \parallel B) = obs(A) \sqcup obs(B)$
3. Pour toute variable x. $obs(\exists_x A) = \exists_x obs(A)$
4. $obs(A + B) = vrai$ $\square$

preuve :
Soit $\tilde{A} = \exists_{x1}...\exists_{xk}(d \parallel A_1 \parallel ... \parallel A_n)$ (resp. $\tilde{B} = \exists_{y1}...\exists_{yl}(e \parallel B_1 \parallel ... \parallel B_m)$), un agent en forme standard tel que $\tilde{A} \equiv A$ (resp. $\tilde{B} \equiv B$). Nous supposons sans perte de généralité que $(\{x_1, ..., x_k\} \cup vc(d)) \cap (\{y_1, ..., y_l\} \cup vc(e)) = \emptyset$
1. immédiat par définition 3.11 et proposition 3.4.
2. il est évident que si $P \equiv P'$ alors $P \parallel Q \equiv P' \parallel Q$.

$$\begin{aligned}
\text{d'où } obs(A \parallel B) &= obs(\tilde{A} \parallel \tilde{B}) \\
&= obs(\exists_{x1}...\exists_{xk}\exists_{y1}...\exists_{yl}(d \parallel A_1 \parallel ... \parallel A_n \parallel e \parallel B_1 \parallel ... \parallel B_m) \\
&= \exists_{x1}...\exists_{xk}\exists_{y1}...\exists_{yl} d \sqcup e \\
&= \exists_{x1}...\exists_{xk} d \sqcup \exists_{y1}...\exists_{yl} e \\
&\qquad (\text{car } \{x_1, ..., x_k\} \cap vc(e) = \emptyset \text{ et } \{y_1, ..., y_l\} \cap vc(d) = \emptyset \\
&= obs(A) \sqcup obs(B)
\end{aligned}$$

$$\begin{aligned}
3.\ obs(\exists_x A) &= obs(\exists_x \tilde{A}) \\
&= \exists_x obs(\tilde{A}) \\
&= \exists_x obs(A)
\end{aligned}$$

4. On a $A + B \equiv vrai \parallel A + B$, donc $obs(A + B) = obs(vrai \parallel A + B)$
De plus $vrai \parallel A + B$ est en forme normale. Donc $obs(vrai \parallel A + B) = vrai$
$\square$

3.2.3 Réduction

Nous définissons dans cette sous section un système de règles d'inférences spécifiant le système de transitions associé à un programme CC. Considérons un programme $P = D.B$.

La réduction utilise deux types de règles : les règles de réduction proprement dites qui décrivent la réduction des agents simples et et les règles structurelles qui construisent la réduction des agents composés et prennent en compte la congruence structurelle.

Règles structurelles :

$$\frac{A \rightarrow A'}{A \parallel B \rightarrow A' \parallel B} \quad \text{(parallel)}$$

$$\frac{A \rightarrow A'}{\exists_x A \rightarrow \exists_x A'} \quad \text{(exists)}$$

$$\frac{A \rightarrow A'}{A + B \rightarrow A'} \quad \text{(choix)}$$

$$\frac{B \equiv A \qquad A \rightarrow A' \qquad A' \equiv B'}{B \rightarrow B'} \quad \text{(équiv)}$$

Règles de réduction :

$$\frac{d \vdash c}{d \parallel ask(c).A \rightarrow d \parallel A} \quad \text{(ask)}$$

$$\frac{p(\vec{x}) :: A \in D \qquad \vec{x} \cap \vec{y} = \emptyset}{p(\vec{y}) \rightarrow \exists_{\vec{x}}(\vec{x} = \vec{y} \parallel A)} \quad \text{(récursion)}$$

Notons que nous avons intégré la propagation de contraintes dans la congruence structurelle. Il peut être intéressant de traiter celle-ci au niveau des règles de réduction par suppression de l'axiome $c \parallel d \equiv d \sqcup c$ et ajout de la règle suivante :

$$\frac{}{c \parallel d \rightarrow c \sqcup d} \quad \text{(propagation de contraintes)}$$

Cela ne permettrait cependant pas de vraiment prendre en compte le fonctionnement du résolveur de contraintes. En effet, les algorithmes utilisés sont d'un fonctionnement beaucoup plus complexe. Une approche intéressante est celle de [BFB$^+$95] dans lequel la sémantique opérationnelle de la PLC est étudiée par rapport à un *critère de rejet* afin de prendre en compte les cas où l'algorihme de satisfiabilité ne repond ni *oui* ni *non*.

Etant donné un programme $P = D.B$, ces règles définissent un système de transitions $T = \langle Agents, \rightarrow \rangle$, où $\rightarrow$ est la plus petite relation sur les agents vérifiant les règles d'inférences définies ci-dessus. Nous l'appelons système de transitions associé à P.

Remarquons que le système de transitions associé à un programme est acyclique. En effet, un éventuel cycle ne pourrait apparaître que par l'intermédiaire de la récursion. Or chaque application de la règle (récursion) introduit un nouvel opérateur de restriction (et celui-ci ne peut pas être éliminé par l'équivalence sructurelle).

La proposition suivante établit une propriété importante des langages CC : la monotonie du store.

Proposition 3.6 L'évolution du store observable est monotone :
Soit $P \rightarrow Q$ une transition, on a $obs(Q) \vdash obs(P)$. □

preuve :
par induction structurelle.
cela est vrai pour les règles de réduction :
(ask) : $obs(d \parallel ask(c).A) = d$ et $obs(d \parallel A) = d \sqcup obs(A)$ (prop. 3.5)
or $\forall c \in c.\ d \sqcup c \vdash d$.
(récursion) : idem que pour l'opérateur de choix car $obs(p(\vec{x})) = vrai$.
règles structurelles :
(parallel) : Si $obs(P') \vdash obs(P)$ alors $obs(P' \parallel Q) \vdash obs(P \parallel Q)$ car
$obs(P' \parallel Q) = obs(P') \sqcup obs(Q)$ (idem pour $P \parallel Q$) (prop. 3.5).
(exists) : Si $obs(P') \vdash obs(P)$ alors $obs(\exists_x P') \vdash obs(\exists_x P)$ car
$obs(\exists_x P') = \exists_x obs(P')$ (idem pour $\exists_x P$) (prop. 3.5).

(choix) : On a $obs(A + B) = vrai$ (prop. 3.5)
donc trivialement, $obs(Q) \vdash obs(P)$ car P est de la forme $A + B$.
(équiv) : immédiat par prop. 3.5.
$\Box$

Définition 3.12 Soit $A_0, A_1, ...$ une dérivation (possiblement infinie) pour A.
L'*observable* de cette dérivation est $\sqcup_i obs(A_i)$.
L'*ensemble des observables* d'un programme $P = D.B$ est l'ensemble des observables des dérivations pour B.
L'ensemble des *observables finis* d'un programme $P = D.B$ est l'ensemble des observables des dérivations finies pour B. $\Box$

Remarquons que les observables finis sont des éléments finis du système de contraintes puisqu'ils sont définis comme étant le plus petit majorant d'un ensemble fini de contraintes finies. L'observable d'une branche infinie peut être un élément fini ou infini du système de contraintes.

3.2.4 Confluence

Nous nous intéressons dans cette sous-section aux propriétés de confluence du système de transitions associé à un programme. Ce système de transitions n'est bien évidement pas confluent de par la règle de réduction (équiv) qui permet d'obtenir, lors d'une transition, n'importe quel représentant d'une classe d'équivalence. Pour des raisons de clarté de présentation, nous avons préféré décrire dans la sous-section précédente la sémantique des langages CC sur les agents. Nous considérons ici la restriction du système de transitions aux classes d'équivalence d'agents. Nous prouvons sa confluence.

Soit P un programme. Nous notons $Agents|_\equiv$ l'ensemble quotient de l'ensemble des agents par la relation de congruence $\equiv$.
Considérons le système de transitions $T|_\equiv = \langle Agents|_\equiv, \rightarrow\rangle$ où $\rightarrow$ est définie de la façon suivante (nous surchargeons le symbole $\rightarrow$) :
$\forall \gamma, \gamma' \in Agents|_\equiv,\ \gamma \rightarrow \gamma'\ ssi\ \exists A \in \gamma, \exists A' \in \gamma'\ tels\ que\ A \rightarrow A'$.

Remarquons que pour toute dérivation $\gamma_0 \rightarrow \gamma_1 \rightarrow ...$ de $T|_\equiv$ il existe une (au moins) dérivation $A_0 \rightarrow A_1 \rightarrow ...$ de T vérifiant : $\forall i,\ A_i \in \gamma_i$ (il peut en exister une infinité, une classe d'équivalence pouvant comporter une infinité de

représentants).
La réciproque est trivialement vraie : pour toute dérivation $A_0, A_1, ...$ de T il existe une dérivation $\gamma_0, \gamma_1, ...$ de $T|_\equiv$ vérifiant : $\forall i,\ A_i \in \gamma_i$.

Définition 3.13 Soit $\gamma \in Agents|_\equiv$.
L'observable $obs(\gamma)$ de γ est l'observable de l'un (quelconque) de ses représentants (rappelons que tous les représentants d'une classe d'équivalence possèdent le même observable (prop. 3.5)).
Soit $\gamma_0 \rightarrow \gamma_1 \rightarrow ...$ une dérivation de $T|_\equiv$.
L'observable de $\gamma_0 \rightarrow \gamma_1 \rightarrow ...$ est $\sqcup_i obs(\gamma_i)$. □

La proposition suivante établit l'équivalence des observables de T et de $T|_\equiv$.

Proposition 3.7 Soit $\gamma_0, \gamma_1, ...$ une dérivation de $T|_\equiv$.
Soit $A_0, A_1, ...$ une dérivation de T vérifiant $\forall i.\ A_i \in \gamma_i$.
Les observables de $\gamma_0, \gamma_1, ...$ et de $A_0, A_1, ...$ sont égaux. □

preuve :
Immédiat par construction des notions d'observables (def. 3.13). □

Nous montrons que le système de transitions $T|_\equiv$ est uniformément confluent pour les programmes CC déterministes. Nous utiliserons les définitions suivantes.

Les agents pouvant être représentés sous forme d'arbres de décomposition (cela est évident de par la grammaire qui définit les agents), nous utiliserons la notion d'occurrence dans les structures d'arbres.

Définition 3.14 Soient A, B deux agents.
$B[A]$ signifie que B contient une (au moins) occurrence de A et réfère à une occurrence particulière. $B[C/A]$ dénote le remplacement dans B de cette occurrence (particulière) de A par C. □

Remarquons que toute transition $P \rightarrow P'$ de T met en jeu une unique règle de réduction.

Définition 3.15 Soit $P \to P'$ une transition de T, soit $S \to S'$ l'unique application d'une règle de réduction qui intervient dans $P \to P'$.
$S \to S'$ est appelée *réduction initiale* (de la transition).
Si $S \to S'$ est de type (choix) ou (récursion) nous appelons *occurrence initiale* l'occurrence de S dans P.
Si $S = d \parallel ask(c).A \to S'$ est de type (ask) alors l'occurrence initiale est celle de $ask(c).A$ dans P. □

Proposition 3.8 (Confluence uniforme)
Soit $P = D.B$ un programme dans lequel l'opérateur de choix n'apparaît pas. Le système de transitions sur les classes d'équivalences d'agents $T|_{\equiv}$ associé à P est uniformément confluent. □

preuve :
Remarquons que si on a $P \to P'$ pour deux agents P et P' avec $S \to T$ pour rèduction initiale, alors il existe $\tilde{P} \equiv P$ tel que :
$\tilde{P}[S] \to \tilde{P}[T/S]$,
$\tilde{P}[T/S] \equiv P'$
et $S \to T$ est la rèduction initiale de $\tilde{P}[S] \to \tilde{P}[T/S]$.
(il suffit pour cela que $\tilde{P}$ soit une forme sandard de P qui conserve le nom des variables de S).
De plus, il est possible d'appliquer un α-renommage à S.
Donc, soient $\gamma, \gamma_1, \gamma_2$, trois noeuds distincts de $T|_{\equiv}$ tels que $\gamma \to \gamma_1$, $\gamma \to \gamma_2$ et $\gamma_1 \neq \gamma_2$.
Il existe les transitions $P[S_1] \to P[T_1/S_1]$ et $P[S_2] \to P[T_2/S_2]$ de T vérifiant $P \in \gamma$, $P[T_1/S_1] \in \gamma_1$ et $P[T_2/S_2] \in \gamma_2$.
Les agents S_1 et S_2 sont disjoints dans P (si ce n'était pas le cas, on aurait $S_1 = S_2$, donc $T_1 = T_2$, d'où $\gamma_1 = \gamma_2$) donc $P[T_1/S_1]$ contient S_2 et $P[T_2/S_2]$ contient S_1.
De plus, S_1 et S_2 sont soit de type $p(\vec{x})$ soit de type $ask(c).A$ (P ne contient pas d'opérateur de choix). La règle de récursion ne nécessite pas de condition à son application. L'évolution du store étant monotone, si la condition d'application d'une règle (ask) est vérifiée pour P, elle l'est pour ses dérivatifs.
Donc nous avons les transitions suivantes :
$P[T_1/S_1][S_2] \to P[T_1/S_1][T_2/S_2]$ et $P[T_2/S_2][S_1] \to P[T_2/S_2][T_1/S_1]$.
$P[T_1/S_1][T_2/S_2] = P[T_2/S_2][T_1/S_1]$ car S_1 et S_2 sont disjoints dans P.
Notons δ la classe d'équivalence de $P[T_1/S_1][T_2/S_2]$ et $P[T_2/S_2][t_1/S_1]$, nous avons $\gamma_1 \to \delta$ et $\gamma_2 \to \delta$. □

Corollaire 3.1 Soit $P = D.B$ un programme dans lequel l'opérateur de choix n'apparaît pas. $T|_{\equiv}$ est confluent et uniforme en compléxité (au sens de la définition 3.8). $\square$

preuve :
Immédiat par les propositions 3.1 et 3.2. $\square$

3.2.5 Non-déterminisme "angélique"

La distinction entre le *non-déterminisme don't know* de la programmation logique et *le non-déterminisme don't care* des langages logiques concurrents se retrouve dans le cadre des langages concurrents avec contraintes sous les appelations *non-déterminisme angélique* [2] et *non-déterminisme démoniaque* [3]. Cette distinction repose sur le traitement sémantique réservé à l'opérateur de choix du point de vue de la définition des observables. Dans le cas angélique on s'intéresse aux observables de toutes les branches issues d'un agent $A + B$ tandis que dans le cas démoniaque on choisit ou bien l'agent A ou bien l'agent B et l'on ne s'intéresse qu'aux observables issus de l'exécution de l'agent choisi. Dans la sémantique que nous avons donné, les observables d'un programme sont l'ensemble des observables de toutes les dérivations. Cette sémantique est donc angélique.

Le non-déterminisme démoniaque pose des problèmes d'un point de vue sémantique. En particulier, en présence d'un opérateur de choix gardé, la confluence n'est assurée que pour une classe réduite de programmes (cf. [FGMP95]). Les langages CC démoniaques sont cependant intéressants de par leur pouvoir d'expression.

Nous prouvons que les observables finis (angéliques) d'un programme sont indépendants du scheduling.

Considérons un programme $P = D.B$ et son système de transitions associé T. La preuve est basée sur le système de transitions sur les classes d'agents $T|_{\equiv}$. Nous définissons au préalable une notion de *scheduling*.

Définition 3.16 Un *scheduling pour* P est une fonction partielle
$s : Agents|_{\equiv} \rightarrow T$

[2] angelic non-determinism
[3] demonic non-determinism

qui a tout état non terminal γ de $T|_\equiv$ associe une transition $R \to R'$ de T vérifiant $R \in \gamma$. $\Box$

L'intuition sous-jacente est de choisir une “occurrence initiale” pour chaque état (classe d'équivalence d'agents) non terminal. Nous effectuons cela en associant à chaque état non terminal une transition de T qui permet de spécifier une occurrence initiale dans un représentant particulier de l'état. Cela est suffisant car si on considère une transition qui réduit un représentant quelconque d'un état, on peut se ramener au représentant particulier utilisé pour définir le scheduling grace à la règle structurelle (équiv).
Une dérivation *satisfait à* un scheduling s si toute transition appartenant à cette dérivation a pour agent initial celui spécifié par s.
De même, la restriction du système de transitions sur les classes d'équivalences d'agents à s est construit en ne considérant que les transitions dont l'agent initial est celui spécifié par s.

Définition 3.17 Soit s un scheduling pour P.
Une dérivation de $T|_\equiv$ *satisfait à* s si toute les transitions $\gamma \to \gamma'$ dont elle est composée vérifient :
Soit $R[S] \to R'$ la transition associée à γ par s, S étant son occurrence initiale.
Il existe une transition $R[S'/S] \to Q$ de T d'occurrence initiale S' telle que $Q \in \gamma'$ et,
Si $S = A + B$ alors $S' = A + B$ ou $S' = B + A$,
Sinon $S' = S$. $\Box$

Définition 3.18 Soit s un scheduling pour P.
$T|_\equiv^s$ est le système de transitions sur les classes d'équivalence d'agents dont la relation de transition est la relation $\to \subseteq Agents|_\equiv \times Agents|_\equiv$ définie par :
$\gamma \to \gamma'$ si et seulement si :
Soit $R[S] \to R'$ la transition associée à γ par s, S étant son occurrence initiale.
Il existe une transition $R[S'/S] \to Q$ de T d'occurrence initiale S' vérifiant $Q \in \gamma'$, avec
Si $S = A + B$ alors $S' = A + B$ ou $S' = B + A$,
Sinon $S' = S$. $\Box$

$T|_\equiv^s$ est appelé *calcul angélique pour s*. Cela correspond intuitivement à la notion d'arbre de recherche de la programmation logique.

Proposition 3.9 $T|^s_\equiv$ est une restriction de $T|_\equiv$. □

preuve : trivial.□

Les observables de $T|^s_\equiv$ sont définis de la même manière que les observables de $T|_\equiv$ (cf. définition 3.13).

Proposition 3.10 Soit s un scheduling pour P.
Les observables finis de $T|^s_\equiv$ et les observables finis de $T|_\equiv$ sont les mêmes. □

(schéma de) preuve :
1. Il est évident que l'ensemble des observables de $T|^s_\equiv$ est inclus dans l'ensemble des observables de $T|_\equiv$.
2. Soit $\gamma_0, \rightarrow \gamma_1 \rightarrow ... \rightarrow \gamma_n$ une dérivation finie de $T|_\equiv$.
Nous définissons une restriction de $T|_\equiv$ dans laquelle le sens de réduction des occurrences d'opérateurs de choix est fixé. La définition formelle d'une telle restriction nécessitant la manipulation d'objets mathématiques syntaxiquement très lourds, nous nous contentons ici de d'une descripion intuitive.
Considérons la restriction $T|^+_\equiv$ de $T|_\equiv$ dans laquelle une même occurrence d'un opérateur de choix est toujours reduite dans le même sens. Ce sens étant choisi arbitrairement sauf pour les occurrences d'opérateurs de choix qui sont réduites dans $\gamma_0, \rightarrow \gamma_1 \rightarrow ... \rightarrow \gamma_n$ auquel cas le sens doit être celui qui est utilisé dans $\gamma_0, \rightarrow \gamma_1 \rightarrow ... \rightarrow \gamma_n$.

La preuve de la proposition 3.8 s'étend aisément aux opérateurs de choix dont le choix est fixé par le scheduling. Donc $T|^+_\equiv$ est confluent.
$\gamma_0, \rightarrow \gamma_1 \rightarrow ... \rightarrow \gamma_n$ appartient à $T|^+_\equiv$ (par construction).
Comme ce dernier est confluent, il existe une dérivation $\gamma_0, \rightarrow \gamma'_1 \rightarrow ... \rightarrow \gamma_n$ de $T|^+_\equiv$ qui satisfait à s.
□

Cette proposition ne s'étend pas aux dérivations infinies. La raison en est la même que pour la non indépendance par rapport à la règle de calcul des branches infinies en programmation logique.
Par exemple, si on considère le programme $D.A$ avec :

$D = p :: p$ et $A = p \parallel x = a$.

On obtiendra un calcul infini dont l'observable est *vrai* avec un scheduling qui choisit toujours l'appel de procédure. Par un scheduling qui choisit $x = a$ (à un moment quelconque du calcul), on obtiendra un calcul infini dont l'observable est $x = a$.

Remarquons cependant que contrairement au cas de la programmation en logique, l'obtention d'un store incohérent n'arrête pas le calcul. La notion d'échec, au sens de la programmation en logique, n'existe pas et l'on ne retrouve pas ici de problème analogue à la non indépendance des branches échec par rapport à la règle de calcul.

3.3 Autres sémantiques

Saraswat définit une sémantique dénotationnelle en terme d'opérateurs de clôture sur l'ensemble des contraintes (cas des determinate CC) ou sur son *powerdomain de Smith* (cf. [Plo83]) (cas des indeterminate CC). Les opérateurs de clôture sont des opérateurs monotones, idempotents et extensifs. Ils bénéficient sur les treillis de la propriété de pouvoir être complètement caractérisés par l'ensemble de leurs points fixes. La sémantique dénotationnelle associe à un programme un opérateur de clôture qui est simplement représenté par un ensemble de contraintes. Cette sémantique est incompatible avec l'opérateur de choix (one step choice) que nous utilisons. Le lecteur intéressé peut se reporter à [SRP91] et [JSS91].

Parmi les sémantiques de la concurrence, la notion de *bisimulation* présente un intérêt tout particulier. Elle permet de prendre en compte le comportement réactif des programmes. Saraswat et Rinard ([SR90]) définissent une telle sémantique pour les CC. Nous reviendrons sur ce sujet au chapitre 6.

3.4 Exemple

Nous donnons un exemple de programme CC adapté de [LM92] qui montre la manière dont s'effectue la "mobilité" des variables, c'est à dire la reconfiguration dynamique du réseau de communication.
Le système de contraintes considéré est celui de l'exemple 2.1 (système de contraintes de Herbrand) sur un ensemble de symboles de fonctions contenant la constante 0 et le constructeur binaire $\langle , \rangle$.
Les variables jouent dans les CC le rôle de canaux de communication. Deux

agents communiquent par l'intermédiaire des variables qu'ils ont en commun.

Considérons l'agent $\exists_y(A \parallel B)$ dans lequel,
$A = (\exists_z y = \langle 0, z\rangle \parallel A')$
$B = (\exists_{y'}\exists_w ask(y = \langle y', w\rangle).(y = \langle y', w\rangle \parallel B'))$

A et B partagent une variable y que nous qualifierons de globale (à A et B). D'une manière informelle, l'agent A crée un lien entre une variable locale z et une variable globale y. L'agent B attend cette variable locale pour continuer le calcul. Après communication, z est devenu "visible" de B par l'égalité entre y et w.
Cela correspond aussi à l'implantation d'une variable impérative par l'intermédiaire d'une liste. Par exemple, $y = \langle 1, \langle 0, y'\rangle\rangle$ est une variable dont la première valeur est 1 et la valeur "actuelle" 0. Elle peut être modifiée en posant une contrainte sur y'. Cette méthode est issue de celle utilisée dans les LLC (cf. [Sha89]). Il est souvent utile, en particulier pour des programmes réactifs, de simuler des variables dont l'évolution est non monotone.

Chapitre 4

Le π-calcul

Issus du CCS ([Mil89]), le π-calcul ([MPW89]) est un modèle de calcul de la concurrence qui suscite un engouement certain. Son interêt réside en sa sémantique et la simplicité de ses opérateurs de base. Ceux-ci reflètent des intuitions simples de mécanismes de communication : envoi ou réception d'un message, et de mécanismes de composition de prosessus : composition parallèle, choix local, déclaration de variable locale. Il est remarquable que ces derniers soient les mêmes que ceux que l'on retrouve dans les langages CC. Le π-calcul et les CC ne diffèrent en fait que par leurs paradigmes de communication et de sychronisation respectifs.

La différence essentielle par rapport au CCS réside dans la possibilité de reconfiguration dynamique du réseau. La communication ne concerne plus seulement la transmission d'un certain type indéterminé d'information mais aussi la transmission de *noms de canaux*.

Le π-calcul, qui ne modélise que la communication à l'exclusion de tout mécanisme de calcul sur des données autres que les canaux de communication eux mêmes, offre cependant un pouvoir d'expression surprenant et permet d'encoder de nombreuses structures de données. Le langage Pict de David Turner et Benjamin Pierce [Pie96] représente une tentative intéressante de construction d'un langage de programmation très évolué avec pour seule base le π-calcul. Il semble cependant difficile, bien que la question reste ouverte, d'encoder les contraintes dans toute leurs généralité (cf. [VP96] dans lequel le γ-calcul ([Smo94]), qui est un langage concurrent avec des contraintes restreintes aux contraintes d'égalité, est encodé dans le π-calcul).

Nous présentons le π-calcul dans sa version asynchrone. Le π-calcul asynchrone à été proposé par Gérard Boudol dans [Bou92]. Notre présentation suit essentiellement [ACS96] à l'exception du fait que l'opérateur de choix définit dans cet article est un opérateur de choix gardé tandis que nous conservons ici celui définit initialement par Milner, Parrow et Walker dans [MPW89].

Afin de faciliter la comparaison avec les langages CC, nous décrivons le π-calcul dans un style proche de celui utilisé dans le chapitre précédent.

4.1 Présentation et syntaxe

Les canaux de communication du π-calcul sont appelés *noms*[1] (noms de canaux). Nous conserverons cette appelation au long de ce chapitre. Considérons un ensemble $\mathcal{N}$ infini dénombrable de *noms*. Nous utiliserons u, v, w comme méta-variables sur les noms.

Le π-calcul asynchrone est un sous-ensemble du π-calcul dans lequel la *sortie* (envoi d'un message sur un nom (canal), le contenu du message étant lui-même un nom) n'est pas un préfixe mais un agent.

Les agents (souvent appellés processus dans les travaux liés au π-calcul) sont définis par la syntaxe suivante :

$$
\begin{array}{lcll}
P & ::= & Dec.A & \text{[programme et agent initial]} \\
Dec & ::= & \epsilon \mid p(\vec{x}) :: A \mid Dec.Dec & \text{[déclarations]} \\
A & ::= & \mathbf{0} \mid \tau.A & \\
 & & \mid \bar{u}v \mid u(v).A & \text{[agents]} \\
 & & \mid A|A \mid A + A & \\
 & & \mid (\nu u)A \mid [u = v]A \mid p(u_1, ..., u_n) &
\end{array}
$$

Comme pour les CC, Un programme est constitué d'un ensemble de déclarations (de procédures) et d'un agent appelé *but* ou *agent initial*.

Etant donné un programme $P = D.B$, pour chaque symbole de procédure p d'arité n apparaissant dans un agent de P, il existe dans D une unique déclaration $p(x_1, ..., x_n) :: A$ vérifiant $fn(A) \subseteq \{x_1, ..., x_n\}$ ($fn(A)$ est l'ensemble des noms libres de l'agent A. cf. définition ci-dessous).

---

[1]names

Nous notons *Agents* l'ensemble des agents définis par cette syntaxe.

Le préfixe $u(v)$ représente l'*entrée*. L'agent $\bar{u}v$ représente la *sortie*. Intuitivement, $\bar{u}v$ est l'envoie du nom v sur le nom (canal) u et $u(v).A$ est la réception d'un nom sur le nom (canal) u. Soit w le nom ainsi reçu, $u(v).A|\bar{u}w$ se réduit en l'agent A dans lequel les occurrences de v sont remplacées par w.
Le préfixe *silencieux* τ dénote une action non visible d'un observateur extérieur au système.
L'opérateur de restriction (νu) est similaire à celui des CC (noté alors $\exists_x$, x étant une variable). u est un nom local à $(\nu u)A$.
Le préfixe de filtrage $[u = v]$ est un opérateur conditionnel : $[u = v]A$ se réduit en A si $u = v$.
$|$ est la composition parallèle et $+$ l'opérateur de choix.

Définition 4.1 Dans un agent de la forme $u(v).A$ ou $(\nu v)A$, l'occurrence du nom v est dite *liante*. Dans les deux cas, la portée de celle-ci est l'agent A.
Une occurrence d'un nom u dans un agent est dite *libre* si et seulement si elle n'est pas dans la portée d'une occcurence liante de u.
Une occurrence d'un nom u dans un agent est dite *liée* si elle n'est pas libre.
Soit A un agent, nous notons $fn(A)$ (resp. $bn(A)$) l'ensemble des noms qui possèdent une occurrence libre (resp. liée) dans A. Nous notons $n(A)$ l'ensemble des noms de A (i.e. $n(A) = fn(A) \cap bn(A)$).
□

Soit A un agent. Soit $[\vec{v}/\vec{u}]$ une substitution sur les noms (cf. def. 2.4). Nous notons $A[\vec{v}/\vec{u}]$ le remplacement simultané dans A de chaque occurrence libre d'un élément de $\vec{u}$ par l'élément correspondant de $\vec{v}$ avec changement des noms liés pour éviter la capture par un opérateur de restriction.

4.2 Sémantique

La sémantique du π-calcul, comme celle du CCS, est définie par un système de transitions annoté décrit dans le style SOS et dont les états sont les agents. Sur ce système de transitions annoté est ensuite construit une relation d'équivalence d'observation appelée bisimilarité. L'idée de base de cette sémantique est qu'observer un processus est la même chose que communiquer avec lui. Les annotations représentent (à l'exception de l'action τ, qui est l'action silencieuse ou non observable) les actions de communication avec l'environnement qu'un processus peut produire. La relation de bisimilarité capture le fait que deux processus peuvent produire les mêmes suites d'actions, et sont donc indifférentiables par un observateur extérieur. Prouver que deux processus sont "interchangeables" est important dans le cadre de la concurrence et tout particulièrement dans celui de la distribution. Considérons par exemple le cas d'un mécanisme de client-serveur comportant plusieurs serveurs concurrents, il est important de pouvoir capturer le fait que deux serveurs offrent le même service aux clients.

Nous présentons la forme la plus simple de bisimulation qui est appelée *bisimulation close*[2] dans ses versions *forte* et *faible*. Nous omettrons l'adjectif *clos* par la suite. Les bisimulations fortes observent toutes les actions. La bisimulation faible n'observe pas les actions silencieuses τ. Cette dernière correspond donc plus précisément à l'intuition que nous voulons capturer, les actions silencieuses étant *locales* et donc sensées être non observables. La notion de bisimulation a été étudiée en profondeur dans la littérature (e.g. [MPW89, BS96] pour le π-calcul synchrone, [ACS96, HT92] pour le π-calcul asynchrone).

Nous présentons le système de transitions annoté dans sa version *early instantiation* (règle (entrée-act)).

4.2.1 Congruence structurelle et système de transitions

Nous définissons tout d'abord les systèmes de transitions annotés.

Définition 4.2 Un *système de transitions annoté* est une structure $\langle \Gamma, A, \rightarrow \rangle$ dans laquelle Γ est un ensemble d'éléments γ appelés *états*, A est un ensemble d'annotations et $\rightarrow \subseteq \Gamma \times A \times \Gamma$ est une relation appelée *relation de transition.* $\square$

Par la suite, nous écrirons $\gamma \xrightarrow{a} \gamma'$ pour $(\gamma, a, \gamma') \in \rightarrow$.
Nous dirons que $\gamma \xrightarrow{a} \gamma'$ est une *transition.*

Les annotations, appelées *actions*, sont d'un des types suivants :

- τ est l'action *silencieuse.* $A \xrightarrow{\tau} B$ signifie que A peut évoluer en B sans aucune interraction avec l'environnement.

[2] ground bisimulation

- $\bar{u}v$ est une action de *sortie libre.* $A \stackrel{\bar{u}v}{\rightarrow} B$ signifie que A peut émettre le nom libre v sur le canal u et évoluer en B.
- uv est une action d'*entrée.* $A \stackrel{uv}{\rightarrow} B$ signifie que A peut recevoir le nom v sur le canal u et évoluer en B.
- $\bar{u}(v)$ est une action de *sortie liée.* $A \stackrel{\bar{u}(v)}{\rightarrow} B$ signifie que A peut émettre le nom local (lié dans A) v sur le canal u et évoluer en B.

L'ensemble des actions est noté Act.

Les noms u et v sont dits *libres* dans les actions $\bar{u}v$ et uv.
u est dit *libre* et v est dit *lié* dans l'action $\bar{u}(v)$.

Soit α une action. Nous notons $fn(\alpha)$ (resp. $bn(\alpha)$) l'ensemble des noms libres (resp. liés) de α, et $n(\alpha)$ l'ensemble des noms de α (i.e. $n(\alpha) = fn(\alpha) \cup bn(\alpha)$).

Notons que l'action d'entrée est une action *libre*: v est libre dans uv (tandis que v est lié dans l'agent $u(v).A$). Cela provient de notre choix d'utiliser un système de transitions avec *early instantiation.*

La congruence structurelle $\equiv$ sur les agents est la plus petite congruence vérifiant les axiomes suivants :

- Si A et B sont α-équivalents (ils ne diffèrent que par le choix des noms liés) alors $A \equiv B$.
- $\langle Agents, |, \mathbf{0} \rangle$ est un monoïde commutatif.
- $\langle Agents, +, \mathbf{0} \rangle$ est un monoïde commutatif.
- $[u = u]A \equiv A$
- $(\nu u)A|B \equiv (\nu u)(A|B)$ si u n'est pas libre dans B
  $(\nu u)A + B \equiv (\nu u)(A + B)$ si u n'est pas libre dans B
  $(\nu u)(\nu y)A \equiv (\nu y)(\nu u)A$
  $(\nu u)A \equiv A$ si u n'est pas libre dans A

Il n'est en fait pas nécessaire, pour la définition d'une sémantique de bisimulation, d'imposer une congruence structurelle aussi forte, certains axiomes (ceux concernant l'opérateur de choix) étant conséquence de la relation de bisimilarité elle-même. Nous conservons ceux-ci afin de faciliter notre propos de comparaison du π-calcul et des CC.

Etant donné un programme $P = D.B$, Le système de transitions annoté associé à P est défini par le système de règles d'inférence du tableau 4.1.

4.2.2 Equivalence d'observation

La notion de simulation du π-calcul synchrone prend en compte l'observation des actions d'entrée et de sortie. Dans le cas du π-calcul asynchrone, le cas des actions de sortie ne pose pas de problème : il paraît évident de devoir les observer. Cela est moins clair en ce qui concerne les actions d'entrée. Il semble même naturel de restreindre l'observation aux actions de sortie, l'intuition étant, selon le principe de Milner qu'observer un processus est équivalent à communiquer avec lui, qu'un observateur n'a aucun moyen de savoir si le message qu'il a envoyé a bien été reçu. Tout ce que l'agent qui envoie un message peut faire est d'introduire un agent *sortie* dans le système. A moins d'avoir programmé un mécanisme explicite d'accusé de reception, il n'a aucun moyen de savoir si son message a été consommé. Afin de quand même prendre en compte les actions d'entrée, nous recherchons non pas simplement une entrée mais une entrée ou une équivalence d'observation en présence d'un consommateur de cette entrée.

Définition 4.3 Une relation binaire sur les agents $\sim$ est une *simulation forte* si :
Soient A et B deux agents. Si $A \sim B$ alors

1. Si $A \xrightarrow{\alpha} A'$, α n'est pas une action d'entrée et $bn(\alpha) \cap fn(B) = \emptyset$ alors il existe B' tel que $B \xrightarrow{\alpha} B'$ et $A' \sim B'$.
2. Si $A \xrightarrow{uv} A'$ alors
   soit il existe B' tel que $B \xrightarrow{uv} B'$ et $A' \sim B'$
   soit il existe B' tel que $B \xrightarrow{\tau} B'$ et $A' \sim (B'|\bar{u}v)$

□

Définition 4.4 Une relation $\sim$ est une *bisimulation forte* si $\sim$ et sa relation inverse sont des simulations fortes.
La relation de *bisimilarité forte* est la plus grande bisimulation forte. □

Tableau 4.1 : règles d'inférence pour le π-calcul

$$\frac{}{\tau.A \xrightarrow{\tau} A} \quad \text{(tau-act)}$$

$$\frac{}{\bar{u}v \xrightarrow{\bar{u}v} \mathbf{0}} \quad \text{(sortie-act)}$$

$$\frac{}{u(v).A \xrightarrow{uw} A[w/v]} \quad \text{(entrée-act)}$$

$$\frac{A \xrightarrow{\alpha} A'}{A + B \xrightarrow{\alpha} A'} \quad \text{(choix)}$$

$$\frac{A \equiv A' \qquad A \xrightarrow{\alpha} B \qquad B \equiv B'}{A' \xrightarrow{\alpha} B'} \quad \text{(équiv)}$$

$$\frac{A \xrightarrow{\alpha} A'}{[u = u]A \xrightarrow{\alpha} A'} \quad \text{(filtrage)}$$

$$\frac{p(\vec{v}) :: A \in D \qquad A[\vec{u}/\vec{v}] \xrightarrow{\alpha} A'}{p(\vec{u}) \xrightarrow{\alpha} A'} \quad \text{(récursion)}$$

$$\frac{A \xrightarrow{\alpha} A' \qquad bn(\alpha) \cap fn(B) = \emptyset}{A|B \xrightarrow{\alpha} A'|B} \quad \text{(parallel)}$$

$$\frac{A \xrightarrow{\bar{u}v} A' \qquad B \xrightarrow{uv} B'}{A|B \xrightarrow{\tau} A'|B'} \quad \text{(communication)}$$

$$\frac{A \xrightarrow{\bar{u}(v)} A' \qquad B \xrightarrow{uv} B' \qquad v \notin fn(B)}{A|B \xrightarrow{\tau} (\nu v)(A'|B')} \quad \text{(fermeture)}$$

$$\frac{A \xrightarrow{\alpha} A' \qquad v \notin n(\alpha)}{(\nu v)A \xrightarrow{\alpha} (\nu v)A'} \quad \text{(restriction)}$$

$$\frac{A \xrightarrow{\bar{u}v} A' \qquad u \neq v}{(\nu v)A \xrightarrow{\bar{u}(v)} A'} \quad \text{(ouverture)}$$

Les versions fortes des simulations ont le défaut d'observer les actions silencieuses qui, étant internes à un agent, devraient justement être non observables. Par exemple les agents $\mathbf{0}$ et $\tau.\mathbf{0}$ ne sont pas fortement bisimilaires, $\tau.\mathbf{0}$ pouvant produire une action τ. Les simulations (et bisimulations) faibles ignorent les actions silencieuses dans la comparaison des transitions. Nous suivons leur présentation standard qui consiste à définir un système de transitions faible dont la relation de transition ne tient pas compte des actions silencieuses.

Notons $\Rightarrow$ la clôture réflexive et transitive de $\xrightarrow{\tau}$.
Notons $\overset{\alpha}{\Rightarrow}$ la relation $\Rightarrow\xrightarrow{\alpha}\Rightarrow$.

Définition 4.5 La relation de transition faible $\overset{\hat{}}{\Rightarrow}$ est la plus petite relation définie par :
Soit A et B deux agents,
Si $\alpha \neq \tau$ et $A \overset{\alpha}{\Rightarrow} B$ alors $A \overset{\hat{\alpha}}{\Rightarrow} B$.
Si $A \Rightarrow B$ alors $A \overset{\hat{\tau}}{\Rightarrow} B$. $\Box$

Les bisimilations et la bisimilarité faibles sont obtenues par remplacement dans la définition 4.3 de $A \xrightarrow{\alpha} B$ par $A \overset{\hat{\alpha}}{\Rightarrow} B$.

Définition 4.6 Une relation binaire sur les agents $\sim$ est une *simulation faible* si :
Soient A et B deux agents. Si $A \sim B$ alors

1. Si $A \overset{\hat{\alpha}}{\Rightarrow} A'$, α n'est pas une action d'entrée et $bn(\alpha) \cap fn(B) = \emptyset$ alors il existe B' tel que $B \overset{\hat{\alpha}}{\Rightarrow} B'$ et $A' \sim B'$.
2. Si $A \overset{\hat{uv}}{\Rightarrow} A'$ alors
   soit il existe B' tel que $B \overset{\hat{uv}}{\Rightarrow} B'$ et $A' \sim B'$
   soit il existe B' tel que $B \overset{\hat{\tau}}{\Rightarrow} B'$ et $A' \sim (B'|\bar{u}v)$

$\Box$

Définition 4.7 Une relation $\sim$ est une *bisimulation faible* si $\sim$ et sa relation inverse sont des simulations faibles.
La relation de *bisimilarité faible* $\dot{\sim}$ est la plus grande bisimulation faible. $\Box$

Les propriétés des bisimulations ont été beaucoup étudiées pour le π-calcul synchrone. Le π-calcul asynchrone est plus récent et à fait l'objet de moins de travaux. En particulier, l'équivalence d'observation que nous avons définit et qui adapte le traitement des actions d'entrée au cas de la communication asynchrone n'a été proposée que très récemment dans [ACS96]. Nous renvoyons le lecteur intéressé à cet article.

Chapitre 5

Langages CC avec communication par messages

Nous présentons dans ce chapitre un nouveau paradigme de calcul concurrent par contraintes – les langages MCC[1] – dont les bases reposent sur des idées issues des langages CC et du π-calcul. Ce paradigme combine d'une manière homogène communication *ask/tell* et communication par messages. D'un point de vue historique, les motivations qui ont présidées à son émergence sont liées au problème de la distribution des langages CC. La présentation des MCC dans ce chapitre suit ces motivations. En particulier, nous justifierons les choix effectués par des considérations concernant la distribution des langages ainsi obtenus.

5.1 Présentation, motivations

Les langages CC bénéficient d'une sémantique claire et d'un pouvoir d'expression attractif. Ils forment un cadre d'un intérêt évident pour calculer de manière concurrente par informations partielles. Cependant, certains de leurs traits essentiels semblent faire obstacle à leur distribution.

La communication et la synchronisation se font par l'intermédiaire d'un *store* global. Certains modèles de calcul distribué sont basés sur l'idée d'une mémoire partagée (par ex. Linda ([CG89])). Il s'agit d'une approche intéressante dont l'effectivité repose sur la possibilité d'implanter de manière distribuée cette mémoire partagée. Or cette condition semble difficile à satisfaire dans le cas des

[1] Message-based CC

CC : d'une part l'opérateur *ask* nécessite de tester la relation de conséquence sur les contraintes de façon globale ; en effet, le fait qu'une contrainte soit conséquence du store dépend à priori de la totalité de l'information contenue dans le store. D'autre part, l'opérateur *tell* considéré par Saraswat dans [Sar89] est ce qu'il appelle le *tell atomique* : une contrainte n'est publiée que si elle est cohérente avec le store, dans le cas contraire l'agent *tell* est *aborté*. Dans un système distribué, cela conduit soit à la résolution distribuée de contraintes soit à la diffusion[2] de toute l'information publiée. La résolution distribuée de contraintes apparaît comme étant un problème difficile : aucun algorithme efficace et assez général n'existe actuellement. Bien que cela soit très coûteux, on pourrait envisager de satisfaire aux problèmes liés au *ask* en maintenant une copie du *store* sur chaque site. Cependant, même en employant cette solution extrème – on aimerait une vraie notion de calcul local permettant de ne pas diffuser la totalité de l'information calculée – le *tell atomique* poserait toujours problème : celui-ci nécessite un test immédiat de cohérence d'avec le store. Il ne suffirait donc pas de diffuser toute l'information publiée, il faudrait de plus que l'exécution d'un *tell atomique* mette à jour de façon atomique les copies du store. Cela est évidemment irréaliste d'un point du vue pratique.

Partant de ce constat, Saraswat propose ([Sar89], Chap. 7) l'*eventual tell* qui est le *tell* sans test de cohérence. Ainsi, dans un système distribué, il n'est plus requis que les contraintes soient immédiatement communiquées à tous les sites : il est suffisant que cela soit effectué en un temps fini. Malheureusement, l'*eventual tell* à pour conséquence de rendre difficile la construction de programmes sans échecs. Comment en effet assurer alors la cohérence du store ? Or cette propriété apparaît souhaitable dans un cadre distribué afin d'assurer la résistance aux pannes ; un système doit ne pas s'arrêter dès qu'un problème survient sur un site. Le langage Janus ([SKL90]) à été créé dans le but de répondre à ce problème. Cependant, les conditions qu'il impose sur le langage et le système de contraintes (en particulier, une variable ne doit avoir qu'un producteur et qu'un consommateur) sont assez restrictives.

Un système distribué se doit de fournir la possibilité d'effectuer du calcul local, ainsi qu'un mécanisme de transmission du résultat. Les langages CC offrent cela par restriction de la portée des variables (opérateur de restriction), ce qui donne lieu à une certaine notion de variable locale ; mais celle-ci n'est vraiment effective qu'au niveau syntaxique. Dans ce cadre, le résultat d'un calcul local ne peut être transmis que par l'intermédiaire du partage des variables,

[2] broadcast

ce qui signifie que ce résultat est inséparable de l'information concernant les variables locales – et de la totalité de cette information, des contraintes portant sur des variables locales pouvant contraindre des variables globales. Il faudrait pouvoir séparer l'information purement locale de l'information significative au niveau global, définir ce que Saraswat appelle des "fire walls" dans la discussion à ce sujet, chapitre 7 de sa thèse. Cependant, décider de ce qu'il est nécessaire de diffuser et de ce qui ne l'est pas risque d'être très difficile en général. Une approche serait d'utiliser des propriétés spécifiques à certains systèmes de contraintes – par exemple des conditions syntaxiques permettant d'avoir des propriétés d'indépendance de variables ; on retrouve d'ailleurs cette idée dans Janus.

Nous pensons que la question de la distribution des langages CC doit être considérée d'un point de vue différent. Nous proposons de remettre en cause certains traits fondamentaux des CC qui concernent la globalité du store, la communication et la synchronisation. Ayant pour objectif l'élaboration d'un langage adapté à la programmation distribuée, nous voudrions disposer de "vraies" notions d'information locale et de calcul local. Il n'est bien-sûr pas très satisfaisant de conserver une copie du store sur chaque site : il est coûteux et inutile de diffuser l'information locale. Nous proposons l'introduction d'une notion de store local, au plein sens du mot local : deux stores locaux n'évoluent pas forcement de façon à être équivalents et peuvent même devenir incohérents entre eux.

5.1.1 Exprimer la distribution potentielle

La plupart des langages de programmation distribuée explicitent complètement la distribution au niveau du langage même. Un programme consiste alors en un ensemble de "programmes locaux" destinés à être exécutés sur des sites à priori différents. On peut s'interroger sur la raison pour laquelle un pas analogue à celui qui consiste à passer du parallélisme à la concurrence n'a pas été franchi pour la distribution. Cela reste un fait que les modèles proposés explicitent la façon dont sera effectuée la distribution. Nous pensons qu'un intermédiaire entre programmation non distribuée et programmation distribuée au sens usuel est nécessaire, l'idée étant alors de proposer un modèle permettant d'exprimer la distribution potentielle.

L'algorithmique distribuée s'est avérée être une problèmatique difficile. Construire des programmes distribués nécessite un type de raisonnement, une compréhension du problème à résoudre, qui ne nous est pas naturel, ou du moins que nous ne maîtrisons pas encore, et les programmes parallèles efficaces ne sont pas directement issus de versions séquentielles. Il semble intéressant de proposer un modèle de calcul qui soit à la distribution l'analogue de ce qu'est la concurrence au parallélisme ; c'est à dire un paradigme déclaratif permettant de spécifier la distribution d'une manière partielle. Plus précisément, il sagit d'exprimer dans le langage la distribution potentielle, l'implantation choisissant alors la distribution "effective" dans le respect de celle-ci.

5.2 MCC : langage et sémantique

Nous proposons un paradigme de calcul concurrent avec contraintes qui intègre communication *ask/tell* et communication par messages. Nous l'appelons MCC (Message-based Concurrent Constraint programming). Cette section définit sa syntaxe et sa sémantique opérationnelle.

5.2.1 Le paradigme de communication

Nous introduisons dans les CC tels qu'ils on été définis au chapitre 3 deux primitives (agents simples) de communication par message que nous notons *send* et *receive*. La communication intervient par l'intermédiaire de canaux de communications. Nous supposons donc l'existence d'un ensemble de noms de canaux de communication que nous appelerons aussi *noms* afin de rester proche de la terminologie propre au π-calcul ; nous reviendrons sur les liens de ce dernier avec les MCC au chapitre 7. Les noms (de canaux) seront en fait modélisés par des variables du système de contraintes. Celui-ci nous permettra de formaliser d'une manière simple et élégante les mécanismes liés à la communication, montrant ainsi – s'il en est encore besoin – le grand intérêt que présente la notion de contrainte en théorie des langages de programmation.

Un programme MCC exprime la distribution potentielle. Ce programme sera dit *distribuable* – dans le sens où il peut être exécuté de manière distribuée – s'il vérifie certaines propriétés. Intuitivement, un programme MCC distribuable est un réseau de sites communiquants par messages par l'intermédiaire de noms (de canaux), chaque site étant un store sur lequel est exécuté un ("morceau" de) programme MCC. Considérons, dans un but explicatif (rappelons qu'au niveau formel variables et noms sont des objets identiques et ne se distinguent que par leur emploi), que les variables du système de contraintes sont partionnées en un ensemble de *variables de contraintes* et un ensemble de noms (que l'on appellera aussi *variables de canaux*). Le cas simple de programme distribuable est celui d'un programme qui est formé de plusieurs *sites* qui ne partagent aucune variable de contraintes (les sites ne partagent donc que des variables de canaux). En effet, ce qui rend difficile la distribution des CC est le fait que l'information soit échangée par l'intermédiaire du partage de variables (de contraintes). Cependant l'absence de partage de variables de contraintes ne suffit pas à assurer l'indépendance opérationelle des sites (au niveau de la résolution de contraintes : il ne faut pas qu'une contrainte sur un site libère un agent $ask(c).A$ sur un autre site, i.e. que c soit conséquence du plus petit majorant des stores locaux.). Nous devons imposer :

1. que le système de contraintes utilisé vérifie pour toutes contraintes c et d :
   Si c est conséquence de d, avec $d \neq faux$, alors $vc(c) \subseteq vc(d)$.
   (rappelons que, c étant une contrainte, $vc(c)$ est l'ensemble des variables contraintes de c (definition 2.2))

Cette condition est nécessaire car elle ne semble pas être conséquence des axiomes de la définition des systèmes de contraintes que nous utilisons (def. 2.1). Il est cependant clair que les systèmes de contraintes définis par une formalisation basée sur un langage du premier ordre (e.g. [JL87] ou [Sar92a]) satisfont à cette propriété. Et c'est aussi le cas de tous les systèmes de contraintes "concrets" habituellement considérés.

2. que l'opérateur ask ne soit pas libéré par la contrainte $faux$. En effet, si une incohérence apparaît sur un site cela ne doit pas libérer les agents ask sur d'autres sites. Il nous faudra donc imposer cela en prémisse de la règle de réduction (ask).

3. Que toute contrainte c qui est argument d'un opérateur ask vérifie la propriété suivante :
   supposons $d \sqcup e \vdash c$ avec $d \not\vdash c$ alors $vc(e) \neq \emptyset$.

Cette propriété permettra d'éviter que, malgré la séparation des variables, une contraintes sur un site dont l'ensemble de variables contraintes est vide ne libère un ask sur un autre site. Notons que cette propriété est très peu restrictive. Si on considère un système de contraintes défini sur un langage du premier ordre, toute contrainte ne comportant pas de quantificateur existentiel la vérifie. De plus, cette propriété est vérifiée par les contraintes du type $\exists_x y = T[x]$ où $T[x]$ est un terme comportant au moins une occurrence de x. Ce type de contrainte permet le passage de paramètre par unification dans le cadre de la communication *ask/tell* comme cela sera mis en évidence dans l'exemple 6.2.4.

Ces propriétés qui se justifient par des considérations de *distribuabilité* ne sont en fait pas nécessaires à la définition des MCC qui forment un paradigme très général de calcul par contraintes avec communication par messages. Elles ne seront utiles que pour les DCC (Distributed CC) que nous définirons dans le chapitre suivant, les DCC étant destinés à la construction de programmes *distribuables* (en un sens précis, voir plus loin). Nous les imposons sur les MCC afin de donner une approche plus homogène des MCC et des DCC. En particulier, cela nous permet de définir les DCC en tant que sous langage des MCC, ce qui ne serait pas possible si la condition 2 n'était pas présente dans les MCC (règle de réduction du *ask*).

Les agents *send* et *receive* prennent en argument le nom de canal sur lequel la communication doit avoir lieu et l'objet informatique qui doit être communiqué. Dans le cadre de programmation par contraintes qui est le nôtre, il paraît évident que cet objet doit lui-même être une contrainte. Cela pose cependant un problème : celui de la signification de la contrainte reçue. En effet, supposons qu'un site A communique à un site B la contrainte $x < 5$. La variable x étant locale au site A, ce que l'on veut communiquer alors est qu'une certaine variable y, locale au site B, est inférieure à 5 ; et ajouter la contrainte $y < 5$ au store du site B. Les objets communiqués ne doivent donc pas être des contraintes mais des "abstractions de contraintes", c'est à dire des contraintes abstraites de (certaines de) leurs variables ; le site récepteur renommant celles-ci par des variables locales (cf. figure 5.1).

Formellement, nous définissons les abstractions de contraintes comme étant des couples formés d'un vecteur linéaire de variables et d'une contrainte. Nous avons besoin de ce vecteur de variables pour, lors de la réception d'un message, remplacer les variables abstraites par les variables locales spécifiées par l'agent de réception. Ce dont nous avons besoin en fait est d'un mécanisme qui établisse une correspondance entre les variables de la contrainte envoyée et celles du site de réception. Nous aurions pu utiliser à cet effet une formalisation différente

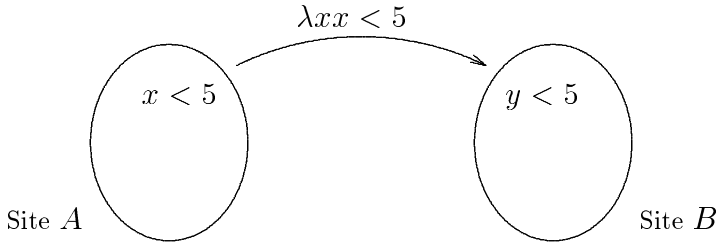

Figure 5.1 : Communication par abstractions de contraintes

qui consisterait en la donnée d'un ordre total sur l'ensemble des variables. Cela permettrait de transmettre des contraintes (et non pas des abstractions de contraintes), le renommage à la réception s'effectuant alors selon cet ordre et l'ordre dans lequel les variables de réception sont spécifiées. La notion d'abstraction de contrainte que nous utilisons définit pour sa part un ordre sur les variables qui est local à chaque message. Une autre idée de formalisation de ce mécanisme serait d'utiliser une notation à la De Bruijn sur des contraintes définies selon un langage du premier ordre, ce qui serait une autre façon de spécifier un ordre local à chaque message.

Considérons un système de contraintes $\mathcal{C}$.

Définition 5.1 L'ensemble des *abstractions de contraintes* de $\mathcal{C}$ est :
$\mathcal{C}_{abs} = \{(\vec{x}, c) | c \in C,\ \vec{x} \subseteq vc(c)\ et\ \vec{x}\ linéaire\}$ □

Nous noterons par la suite $\lambda\vec{x}c$ l'abstraction de contrainte $(\vec{x}, c)$.

L'application d'un vecteur de variables à une abstraction de contrainte est la contrainte obtenue par substitution des variables abstraites :

Définition 5.2 Soit $\lambda\vec{x}c$ une abstraction de contrainte. Soit $\vec{y}$ un vecteur de variables de même longueur que $\vec{x}$.
L'*application de $\vec{y}$ à $\lambda\vec{x}c$* est la contrainte $c[\vec{y}/\vec{x}]$.
Si $\vec{y}$ n'est pas de même longueur que $\vec{x}$ alors l'application de $\vec{y}$ à $\lambda\vec{x}c$ est indéfinie.
□

Proposition 5.1 Soient $\lambda\vec{x}c$ et $\lambda\vec{y}d$ deux abstractions de contraintes vérifiant $long(\vec{x}) = long(\vec{y}) = n$.

Soit $\vec{z}$ un vecteur linéaire de variables de longueur n vérifiant
$\vec{z} \cap (vc(c) \cup vc(d)) = \emptyset$.
Soit $\vec{w}$ un vecteur de variables de longueur n.
On a : si $c[\vec{z}/\vec{x}] = d[\vec{z}/\vec{y}]$ alors $c[\vec{w}/\vec{x}] = d[\vec{w}/\vec{y}]$ □

preuve :
Supposons $c[\vec{z}/\vec{x}] = d[\vec{z}/\vec{y}]$.
On a $c[\vec{w}/\vec{x}] = (c[\vec{z}/\vec{x}])[\vec{w}/\vec{z}]$ et $d[\vec{w}/\vec{y}] = (d[\vec{z}/\vec{y}])[\vec{w}/\vec{z}]$ (cf. chap. 2).
d'où le résultat. □

La relation d'équivalence suivante sur les abstractions de contraintes capture le fait que deux abstractions de contraintes sont équivalentes par application.

Définition 5.3 $\approx^{\mathcal{C}} : \mathcal{C}_{abs} \times \mathcal{C}_{abs}$ est la relation sur les abstractions de contraintes définie par :
$\lambda\vec{x}c \approx^{\mathcal{C}} \lambda\vec{y}d$ si et seulement si :
$long(\vec{x}) = long(\vec{y}) = n$ et,
soit $\vec{z}$ un vecteur linéaire de variables de longueur n tel que $\vec{z} \cap (vc(c) \cup vc(d)) = \emptyset$,
$c[\vec{z}/\vec{x}] = d[\vec{z}/\vec{y}]$. □

La proposition 5.1 établit que le choix du vecteur $\vec{z}$ est sans importance.

Nous noterons cette relation $\approx$ lorsque cela ne génère pas d'ambiguité. C'est-à-dire toujours, puisque afin de simplifier le formalisme, nous définissons les MCC par rapport à un unique système de contraintes.

La remarque suivante est triviale :
Soit $\lambda\vec{x}c$ une abstraction de contrainte. Soit $\vec{y}$ un vecteur linéaire de variables tel que $long(\vec{y}) = long(\vec{x})$ et $\vec{y} \cap vc(c) = \emptyset$.
$\lambda\vec{x}c \approx \lambda\vec{y}c[\vec{y}/\vec{x}]$.

Proposition 5.2 $\approx$ est une relation d'équivalence. □

preuve :
Soient $\lambda\vec{x}c$, $\lambda\vec{y}d$ et $\lambda\vec{w}e$ trois abstractions de contraintes.
1. réflexivité : trivial.

2. symétrie : trivial par symétrie de l'égalité.
3. transitivité : Si $c[\vec{z}/\vec{x}] = d[\vec{z}/\vec{y}]$ et $d[\vec{z}'/\vec{y}] = e[\vec{z}'/\vec{w}]$,
nous avons par proposition 5.1 $d[\vec{z}/\vec{y}] = e[\vec{z}/\vec{w}]$
d'où $c[\vec{z}/\vec{x}] = e[\vec{z}/\vec{w}]$.
□

La relation $\approx$ est compatible avec l'application. La proposition suivante découle directement de la proposition 5.1.

Proposition 5.3 Soient $\lambda\vec{x}c$ et $\lambda\vec{y}d$ deux abstractions de contraintes vérifiant $\lambda\vec{x}c \approx \lambda\vec{y}d$.
Pour tout $\vec{z} \subseteq V$ on a : l'application de $\vec{z}$ à $\lambda\vec{x}c$ est égale à l'application de $\vec{z}$ à $\lambda\vec{y}d$. □

preuve :
Soit $n = long(\vec{x})$ $(= long(\vec{y}))$.
Si $long(z) \neq n$ alors l'application à $\lambda\vec{x}c$ et $\lambda\vec{y}d$ est indéfinie.
Si $long(z) = n$ alors le résultat découle de la proposition 5.1.
□

Nous définissons les MCC en utilisant l'ensemble quotient de l'ensemble des abstractions de contraintes par la relation $\approx$.

5.2.2 Syntaxe

Considérons un système de contraintes $\mathcal{C}$ vérifiant : $\forall\ c, d \in \mathcal{C}$ telles que $d \vdash c$ et $d \neq faux$, si $x \in vc(c)$ alors $x \in vc(d)$.

Les agents MCC sur $\mathcal{C}$ sont définis par la syntaxe suivante.

$$
\begin{array}{lll}
P & ::= & Dec.A \qquad\qquad\qquad\qquad \text{[programme et agent initial]} \\
Dec & ::= & \epsilon \mid p(\vec{x}) :: A \mid Dec.Dec \qquad\qquad \text{[déclarations]} \\
A & ::= & c \mid ask(d).A \\
 & & \mid send(x, \lambda\vec{y}c) \mid receive(x, \vec{y}).A \qquad \text{[agents]} \\
 & & \mid A \parallel A \mid A + A \mid p(\vec{x}) \mid \exists_x A
\end{array}
$$

c est une contrainte finie, d est une contrainte vérifiant la condition 3 décrite page 81, x est une variable, $\vec{x}$ est un vecteur de variables, $\vec{y}$ est un vecteur linéaire de variables, $\lambda\vec{y}c$ est une classe d'équivalence d'abstractions de contraintes, p est un symbole de procédure appartenant à un ensemble dénombrable de symboles de procédures auquel est associé une fonction d'arité.

Un programme est constitué d'un ensemble de déclarations (de procédures) et d'un agent appelé *but* ou *agent initial*.

Etant donné un programme $P = D.B$, pour chaque symbole de procédure p d'arité n apparaissant dans un agent de P, il existe dans D une unique déclaration $p(x_1, ..., x_n) :: A$.

Les agents *send* et *receive* sont *appelés agents de communication*. Ce sont des *agents simples*. Soit A un agent de communication, sont premier argument est appelé *canal de A*.

Les autres agents et constructeurs d'agents correspondent à ceux des CC.

Cette syntaxe est très générale. En particulier, à l'instar de la syntaxe des *CC* et contrairement à celle de la programmation en logique, elle permet l'existence de variables globales. Bien que cela soit d'un intérêt très limité – et même non souhaitable – dans le cadre d'un langage de programmation non distribué, ne serait-ce que pour des raisons de méthodologie de programmation et de risque d'erreur, il nous apparaît important de conserver cette possibilité pour des programmes distribués. Par exemple, un serveur pourra posséder un canal de communication global par l'intermédiaire duquel les clients poseront leurs requêtes.

Etant donné un système de contraintes $\mathcal{C}$ et un ensemble de symboles de procédures Π, nous notons $Agents^{\mathcal{C},\Pi}$ l'ensemble des agents défini par cette syntaxe. Nous noterons celui-ci $Agents$ lorsque cela ne génère pas d'ambiguïté.

Nous appellerons *agents MCC* les agents définis par cette syntaxe lorsqu'il y a risque de confusion avec les agents (CC) définis au chapitre 3.

5.3 Sémantique opérationnelle

Nous décrivons la sémantique opérationnelle des MCC dans le même style que celui utilisé au chapitre 3. Le concept de "molécule flottante" de la Machine

Abstraite Chimique ([BB92]) est bien adapté à la communication par messages dont les règles de réduction nécessitent la juxtaposition syntaxique de plusieurs agents simples.

La congruence structurelle $\equiv$ sur les agents est la plus petite relation de congruence vérifiant les axiomes suivants :

- $\langle Agents, \|, vrai\rangle$ et $\langle Agents, +, vrai\rangle$ sont des monoïdes commutatifs.
- $\exists_x A \equiv \exists_y A[y/x]$ si y n'est pas libre dans A
  $\exists_x A \parallel B \equiv \exists_x(A \parallel B)$ si x n'est pas libre dans B
  $\exists_x A + B \equiv \exists_x(A + B)$ si x n'est pas libre dans B
  $\exists_x \exists_y A \equiv \exists_y \exists_x A$
  $\exists_x A \equiv A$ si x n'est pas libre dans A
- $c \parallel d \equiv d \sqcup c$

Les notions d'agent en forme standard, de store et d'observable (store observable) d'un agent CC de la sous-section 3.2.2 sont étendues aux agents MCC, ainsi que les propositions énoncées dans cette sous-section.

Définition 5.4 Un agent A est en *forme standard* si il est de la forme :
$$A = \exists_{x1} ... \exists_{xk}(d \parallel A_1 \parallel ... \parallel A_n)$$
où d est une contrainte et les A_i sont :
soit des agents simples ask, appels de procédure ou agents de communication (i.e. ce ne sont pas des contraintes),
soit des agents ΣB_i. $\square$

Proposition 5.4 Pour tout agent, il existe un agent en forme standard structurellement équivalent. $\square$

preuve : par induction structurelle. $\square$

Les notions de *store* et de *store observable* sont identiques à celles définies sur les CC. Nous ne redonnons pas ici ces définitions. Le lecteur est invité à se reporter aux définitions 3.10 et 3.11. Les propositions 3.4 et 3.5 s'étendent aisément aux agents MCC.

5.3.1 Règles de réduction

les règles (parallel), (exists), (choix), (équiv) et (récursion) sont celles de la sous-section 3.2.3.

Nouvelles règles :

$$\frac{d \vdash c \qquad d \neq faux}{d \parallel ask(c).A \rightarrow d \parallel A} \qquad \text{(ask)}$$

$$\frac{d \vdash u = v \qquad d \neq faux \qquad long(\vec{x}) = long(\vec{y})}{d \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).A \rightarrow d \parallel c[\vec{y}/\vec{x}] \parallel A} \qquad \text{(communication)}$$

La régle (communication) réduit la composition parallèle de deux agents *send* et *receive* en présence d'une contrainte cohérente qui établit l'égalité de leurs canaux de communication. C'est-à-dire lorsque l'égalité de leurs canaux de communication est conséquence du store.

La règle (ask) est modifiée par rapport à celle du chapitre 3 : il est ici demandé que le store soit cohérent. La raison du renforcement des préconditions de cette règle est que les MCC sont définis, au niveau théorique, avec un store global. Ce que nous appelons store local étant une partie du store (global) dont l'ensemble des variables (de contraintes) est disjoint de celui du restant du store, et la notion de site correspondant à un tel store local et un programme qui s'exécute sur celui-ci. Dans un cadre distribué, l'incohérence d'un store local sur un site ne doit pas débloquer des agents *ask* sur les autres sites. Cette version de l'opérateur *ask* correspond au *ask atomique* proposé dans [BKPR93].

Remarquons que le *tell* n'est pas un agent primitif. $tell(c)$ peut être implanté par $\exists_y(send(y, \lambda\vec{x}c) \parallel receive(y, \vec{x}).vrai)$ dans lequel $\vec{x} = vc(c)$. Nous avons en effet la dérivation suivante (car $y = y$ est égal à $vrai$) :
$\exists_y(send(y, \lambda\vec{x}c) \parallel receive(y, \vec{x}).vrai) \rightarrow c$.
La variable y étant locale à cet agent, si celui-ci est placé en composition parallèle avec d'autres agents, où même inséré dans un agent avec un “trou”, il ne peut pas y avoir de communication avec d'autres agents.

Etant donné $P = D.B$ un programme MCC. Ces règles définissent un système de transitions $T = \langle Agents, \rightarrow\rangle$, où $\rightarrow$ est la plus petite relation sur les agents vérifiant les règles d'inférences définies ci-dessus. Nous l'appelons système de transitions associé à P.

Proposition 5.5 L'évolution du store observable est monotone. □

preuve :
La preuve de la proposition 3.6 s'étend au cas des MCC. Il suffit de montrer la monotonie de la règle (communication) :
Soit la transition $d \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).A \rightarrow d \parallel c[\vec{y}/\vec{x}] \parallel A$.
Nous avons $obs(d \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).A) = d$
et $obs(d \parallel c[\vec{y}/\vec{x}] \parallel A) = d \sqcup c[\vec{y}/\vec{x}]$.
Nous avons bien $d \sqcup c[\vec{y}/\vec{x}] \vdash d$.
Les modifications apportées aux règles (ask) et (choix) ne posent pas de problème (nous ne détaillons pas les modifications engendrées sur la preuve). □

Les propriétés de confluence des CC ne sont bien sûr pas vérifiées ici à cause du non déterminisme de la communication par messages. Par exemple l'agent $A = send(u, \lambda x x < 5) \parallel receive(u, y).true \parallel receive(u, z).true$ se reduit en $y < 5 \parallel receive(u, z).true$ ou bien en $z < 5 \parallel receive(u, y).true$, ces deux agents étant des états terminaux du système de transitions associé au programme $P = \epsilon.A$.

5.4 Distribuabilité

Nous étudions dans cette section la potentialité de distribution que possèdent les programmes MCC. C'est-à-dire, comment et en combien de sites est-il possible de distribuer un agent MCC.

Les canaux ont été définis comme étant des variables du système de contraintes. En effet, il n'est pas nécessaire d'établir la distinction entre canaux et variables de contraintes au niveau du langage. En toute généralité, il peut même être intéressant que la communication par messages dépende de contraintes complexes (et pas seulement d'égalités entre canaux). Cela peut donner lieu à un style de programmation dirigé par les données de manière très effective.

Dans un cadre distribué, nous voudrions limiter au maximum – voir même totalement éviter – la résolution distribuée de contraintes. Ceci nous conduit à nous intéresser à une classe particulière de programmes : les programmes *distribuables*. Intuitivement, un programme est distribuable s'il est une composition parallèle d'agents pouvant s'exécuter indépendamment les uns des autres sauf en ce qui concerne la communication par messages.

Considérons un programme $P = D.B$.
Un agent A est dit être un *agent pour* P si les symboles de procédures apparaîssant dans A sont définis dans D.

Définition 5.5 Un agent A pour P est *distribuable en A' et A"* (par rapport à P) s'il est structurellement équivalent à un agent de la forme
$\exists_{\vec{x}}(A' \parallel A'')$
et si pour toute réduction $A \rightarrow B$ on a :

- Soit il existe B' tel que $A' \rightarrow B'$ et B est distribuable en B' et A''.
- Soit il existe B'' tel que $A'' \rightarrow B''$ et B est distribuable en A' et B''.
- Soit $A' \equiv (send(u, \lambda\vec{y}c) \parallel \tilde{A}')$, $A'' \equiv (receive(v, \vec{z}).\hat{A}'' \parallel \tilde{A}'')$, A se réduit en $\exists_{\vec{x}}(\tilde{A}' \parallel c[\vec{z}/\vec{y}] \parallel \hat{A}'' \parallel \tilde{A}'')$, et B est distribuable en $\tilde{A}'$ et $(c[\vec{z}/\vec{y}] \parallel \hat{A}'' \parallel \tilde{A}'')$.
- Soit $A'' \equiv (send(u, \lambda\vec{y}c) \parallel \tilde{A}'')$, $A' \equiv (receive(v, \vec{z}).\hat{A}' \parallel \tilde{A}')$, A se réduit en $\exists_{\vec{x}}(c[\vec{z}/\vec{y}] \parallel \hat{A}' \parallel \tilde{A}' \parallel \tilde{A}'')$ et B est distribuable en $(c[\vec{z}/\vec{y}] \parallel \hat{A}' \parallel \tilde{A}')$ et $\tilde{A}''$.

□

Les deux premiers points concernent, en particulier, les transitions (ask) qui sont les seules (avec les transitions (communication)) a nécessiter en prérequis des propriétés de la relation de conséquence sur les contraintes. Intuitivement, pour un agent distribuable, les transitions (ask) n'ont pas besoin d'information extérieure au store local dans lequel elles s'exécutent. En conséquence, si on distribue un tel agent, il ne sera pas nécessaire de recourir à la résolution distribuée de contraintes, celle-ci restant locale. Cela n'est en fait pas tout a fait vrai car les transitions (communication) peuvent avoir pour prérequis des propriétés sur des contraintes d'égalité entre variables (qui sont alors des noms de canaux). La résolution distribuée de contraintes est alors nécessaire mais elle n'intervient que sur des cas très simples (égalités entre variables).

Remarquons que tout agent A est distribuable en A et $vrai$. De plus, tout agent $c \parallel A$ est distribuable en c et $c \parallel A$ puisque $c \parallel A \equiv c \parallel c \parallel A$ (car $c = c \sqcup c$).
Cela motive la définition suivante.

Définition 5.6 Un agent A pour P est *distribuable* (par rapport à P) s'il existe deux agents A' et A'' qui ne sont pas des contraintes et qui vérifient : A est distribuable en A' et A''. □

Proposition 5.6 L'opérateur de restriction respecte la propriété de distribuabilité :
Si A est distribuable en B et C alors $\exists_x A$ est distribuable en B et C. □

preuve :
Si $A \equiv \exists_{\vec{y}}(B \parallel C)$ alors $\exists_x A \equiv \exists_x \exists_{\vec{y}}(B \parallel C)$ (car $\equiv$ est une congruence).
De plus, les dérivations pour A sont "isomorphes" aux dérivations pour $\exists_x A$ (cela est évident de par la règle structurelle (exists)).
□

Par contre la composition parallèle et l'opérateur de choix ne respectent pas la distribuabilité.
Considérons par exemple l'agent $A = (ask(x = a).(y = b) \parallel ask(y = b).B)$ qui est distribuable en $ask(x = a).(y = b)$ et $ask(y = b).B$ puisqu'il n'admet aucune transition. $A \parallel (x = a)$ n'est pas distribuable.
Soit A un agent distribuable, $A + A'$ n'est pas distribuable puisqu'il n'existe pas d'agents B et C (si tous deux sont différents de $vrai$) tels que $A + D \equiv \exists_{\vec{x}}(B \parallel C)$.

La proposition suivante établit une forme d'associativité pour la propriété de distribabilité.

Proposition 5.7 Soit A un agent distribuable en B et C.
Si B est distribuable en B' et B'' alors A est distribuable en B' et $B'' \parallel C$.
De plus $B'' \parallel C$ est distribuable en B'' et C. □

Preuve :
$A \equiv \exists_{\vec{x}}(B \parallel C)$ et $B \equiv \exists_{\vec{y}}(B' \parallel B'')$.
Par renommage approprié de $\vec{y}$ en $\vec{z}$, nous avons :
$A \equiv \exists_{\vec{x}}\exists_{\vec{z}}(B'[\vec{z}/\vec{y}] \parallel B''[\vec{z}/\vec{y}] \parallel C)$.
Le système de transitions vérifie de manière évidente les propriétés nécessaires.
$\square$

La forme de commutativité suivante est triviale (de par la commutativité du $\parallel$) :

Proposition 5.8 Soient A, B, C trois agents.
Si A est distribuable en B et C alors A est distribuable en C et B. $\square$

Ces propriétés nous permettront de parler d'agents distribuables en n agents, n pouvant être supérieur à 2.

Définition 5.7 Un agent A est *distribuable en* $A_1, ..., A_n$ (par rapport à P) si pour tout $i \in 1..(n-1)$ on a :
$A_i, A_{i+1}, ..., A_n$ est distribuable en A_i et $(A_{i+1} \parallel A_{i+2} \parallel ... \parallel A_n)$ $\square$

Remarquons que dans un tel cas nous avons pour tout $i \in 1..n$:
A est distribuable en A_i et $(A_1 \parallel ... \parallel A_i - 1 \parallel A_i + 1 \parallel ... \parallel A_n)$.
La preuve découle directement des deux propositions précédentes.

5.4.1 Caractériser la distribuabilité

La propriété de distribuabilité est bien sûr non décidable. Se pose alors le problème de caractériser une (large) classe de programmes distribuables. L'interprétation abstraite, qui possède de nombreuses applications en Programmation en Logique et dans les domaines qui en sont issus peut permettre d'une part, de prouver distribuables des programmes plus complexes que ceux répondant à notre caractérisation syntaxique (cf. chapitre suivant), et d'autre part, de réaliser une analyse de distribuabilité lors de la compilation ou même de l'exécution et ainsi décider dynamiquement de la distribution effective.

L'interprétation abstraite dans les CC est cependant un domaine de recherche encore ouvert. Il n'existe pas encore de cadre général satisfaisant qui permette une telle analyse. La difficulté majeure par rapport à la PL ou la PLC

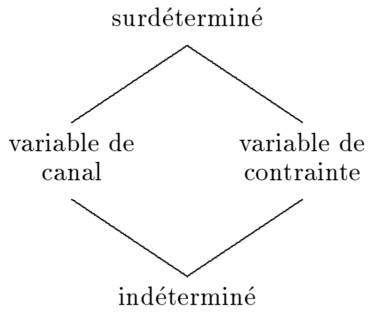

Figure 5.2 : Domaine abstrait

étant que l'on doit également modéliser le flot de contrôle de l'exécution du fait des agents ask.

L'idée serait d'avoir un domaine abstrait permettant de typer les variables et de distinguer variables de canaux et variables de contraintes. Par exemple, en utilisant un treillis (figure 5.2) dans lequel les variables de type $\top$ (variables surdéterminées) seraient potentiellement utilisées en tant que canaux et dans des contraintes "complexes" (autres que des égalités entre canaux). Il s'agirait de variables "à problème" empèchant la distribuabilité. En fait, on ferait alors une approximation de l'exécution qui ne donnerait bien sûr qu'une condition suffisante.

5.5 Opérateurs sur les stores

Le langage que nous avons défini impose à l'utilisateur la complète spécification des contraintes à communiquer. En pratique, un système distribué aura besoin de communiquer des "parties" de stores locaux plutôt que des contraintes. L'exemple typique en est la communication du résultat d'un calcul local exécuté sur un site particulier. Dans un tel cas, ce qui est envoyé est toujours une abstraction de contrainte – un store ou une "partie" d'un store est une contrainte – mais celle-ci ne peut-être directement exprimée dans le langage précédemment décrit car elle dépend du store calculé. L'outil qui nous proposons ci-dessous répond à ce besoin. Nous donnons, en particulier, la possibilité d'utiliser dans un agent *send* non plus seulement des contraintes mais aussi des *opérateurs de store*.

Définition 5.8 Un *opérateur de store* est une fonction $\theta : \vec{v} \times c \mapsto d$,
c, d étant des contraintes et $\vec{v}$ un vecteur de variables, vérifiant :
si $\theta(\vec{v}, c) = d$ alors $vc(d) \subseteq \vec{v}$. $\square$

Nous ne discutons pas dans cette thèse quels opérateurs seraient utiles. Le plus simple d'entre eux – et son intérêt pratique est évident – est sans doute l'opérateur de projection.

Définition 5.9 Soit c une contrainte.
La projection de c sur un vecteur de variables $\vec{x}$ est :
$$proj(\vec{x}, c) \stackrel{def}{=} \exists_{vc(c)/\vec{x}} c \ \square$$

Il est évident qu'un opérateur de projection est un opérateur de store.

Nous introduisons un nouveau type d'agent simple appelé *lien* et noté $lien((\theta, \vec{x}), \vec{y})$. Un tel agent établit un "lien" entre deux ensembles de variables : l'information produite sur les variables $\vec{x}$ est transmise aux variables $\vec{y}$ par l'intermédiaire de l'opérateur θ. Nous parlons ici de *lien* car une fois introduit, un agent *lien* ne peut être retiré et crèe un sorte de "passerelle" entre variables de différents sites (on peut cependant utiliser des liens entre variables d'un même site). La sémantique des agents *lien* est décrite par une règle qui propage l'information disponible sur les variables $\vec{x}$ aux variables $\vec{y}$. Un lien peut être introduit soit directement comme agent dans un programme, soit par l'intermédiaire de la communication par messages. Nous étendons à cet effet la communication par messages aux opérateurs de store.

La syntaxe des langages MCC est étendue avec les agents simples suivants :
$send(u, (\theta, \vec{x}))$ pour θ opérateur de store et $\vec{x}$ vecteur de variables,
$lien((\theta, \vec{x}), \vec{y})$ pour θ opérateur de store et $\vec{x}, \vec{y}$ vecteurs de variables de même longueur.

La communication d'un agent $send(u, (\theta, \vec{x}))$ avec un agent *receive* (cf. ci-dessous) met en place un lien.

La sémantique de ces nouveaux agents est donnée par les règles de réduction suivantes.

$$\frac{d \neq faux}{\begin{array}{c} d \parallel lien((\theta, \vec{x}), \vec{y}) \\ \longrightarrow \\ d \parallel lien((\theta, \vec{x}), \vec{y}) \parallel \theta(\vec{x}, d)[\vec{y}/\vec{x}] \end{array}} \qquad (\text{propagation}_{lien})$$

$$\frac{d \vdash u = v \qquad d \neq faux \qquad long(\vec{x}) = long(\vec{y})}{\begin{array}{c} d \parallel send(u, (\theta, \vec{x})) \parallel receive(v, \vec{y}).B \\ \longrightarrow \\ d \parallel lien((\theta, \vec{x}), \vec{y}) \parallel B \end{array}} \qquad (\text{communication}_{lien})$$

La règle ($\text{communication}_{lien}$) décrit l'introduction d'un agent *lien* par l'intermédiaire de la communication par messages.

La règle ($\text{propagation}_{lien}$) représente un mécanisme opérationnel simple : la propagation du résultat d'un calcul local. Cette règle a le défaut d'introduire des dérivations infinies. Nous avons même : si la règle ($\text{propagation}_{store}$) s'applique à un agent A alors il existe une dérivation infinie pour A (on peut appliquer cette même règle indéfiniment). Cela ne pose pas de problème d'un point de vue pratique (le mécanisme opérationel ainsi simulé est évident). Cela pose cependant un problème sémantique. L'extension de la notion d'observable aux programmes intégrant les agents de communication avec opérateurs de store nécessiterait une notion d'équité.

Une solution à ce problème pourrait être apportée par la définition d'une sémantique opérationnelle dont les états soient des couples formés d'un agent et d'un store comme dans, par exemple, [SRP91]. Cela compliquerait cependant la description de la communication par message qui nécessite la juxtaposition syntaxique des agents *send* et *receive*.

Notons aussi que l'introduction des agents *lien* nécessite de redéfinir les notions de store et de store observable. Plusieurs agents *lien* peuvent créer des liens "croisés" et même des "boucles" entre différents sites. Considérons par exemple l'agent $A \parallel B$, avec $A = \exists_{x1,x2} lien((proj, x1), y1) \parallel x1 = x2$ et $B = \exists_{y1,y2} lien((proj, y2, x2) \parallel y2 = 3$.
Le store attendu $x1 = x2 = y1 = y2 = 3$ peut se calculer par l'intermédiaire d'un opérateur de points fixes.
Considérons une famille finie $L = \{lien(\theta_i, \vec{x}_i, \vec{y}_i)\}_{i \in I}$ d'agents *lien* et l'opérateur $T_L : \mathcal{C} \to \mathcal{C}$ définit par $T_L(c) = c \sqcup (\sqcup_I \theta_i(\vec{x}_i, c)[\vec{y}_i/\vec{x}_i])$. Cet opérateur est

monotone et continu sur le treillis complet formé par l'ensemble des contraintes et la relation de conséquence. Soit A un agent, Soit L la famille (un ensemble dénombrable peut être transformé en une famille, il suffit de fixer une indexation. L'indexation choisie est ici sans importance.) formée par les agents *lien* en composition parallèle de premier niveau dans A. Le "store attendu" de A est $T_L(store(A))$.

Ces nouveaux agents reposent sur des mécanismes opérationnels simples et intuitifs, l'introduction d'un *lien* correspond en quelque sorte à la déclaration d'un calcul local. La communication par messages d'opérateurs de store permet la création dynamique de liens.

La section 6.2 présente des exemples utilisant des agents *lien*.

5.6 Programmation transformationnelle

Les applications les plus évidentes d'un langage distribué concernent les systèmes réactifs et intéractifs. L'utilisation des MCC pour modéliser des problèmes transformationnels peut néanmoins s'avérer intéressante, la communication entre sites étant alors considérée à l'échelle locale (réseaux locaux) voir même au niveau de la coopération entre différents processeurs d'une même machine. Les programmes dont la communication par messages est déterministe correspondent typiquement à ce type d'applications. D'autre part, comme dans le cas des langages CC, il peut être intéressant d'implanter les MCC sur une unique machine, et même de manière séquentielle. Les MCC introduisent ainsi un nouveau style de programmation.

Nous ne développons pas ce sujet dans le cadre de cette thèse. Remarquons cependant qu'il est possible de définir un sous langage des MCC dont les programmes sont à communication déterministes.

5.7 Programmation en MCC : perspectives

La très grande liberté d'utilisation des variables de canaux peut donner lieu à des styles de programmation très divers dont l'intérêt reste à discuter. En particulier, il est possible de faire dépendre la communication par messages de contraintes “complexes” autres que des contraintes d'égalité entre variables de canaux. Cela serait bien sûr un obstacle à la distribuabilité mais on pourrait peut-être déboucher ainsi sur un style de programmation dirigé par les données d'un manière très effective. Nous ne dévellopons pas plus avant ce sujet dans le cadre de cette thèse.

Un intérêt immédiat des MCC est cependant la modélisation de systèmes distribués que nous abordons dans le chapitre suivant par l'intermédiaire d'un sous-langage des MCC.

Chapitre 6

DCC : un sous-langage pour exprimer la distribution

La propriété de distribuabilié est, en toute généralité, complexe à analyser, et s'assurer de la distribuabilité d'un programme MCC est non trivial. Nous développons dans ce chapitre un sous-langage des MCC permettant la construction de programmes MCC distribuables. L'idée de base est que l'absence de contraintes (autres que des égalités) sur les variables utilisées commes canaux de communication permet l'existence d'un "découpage" en *sites* ne communiquant que par messages. nous allons utiliser un typage des variables. En fait nous allons maintenant distinguer deux sortes de variables : les canaux (ou variables de canaux) et les variables de contraintes. Afin de bien distinguer entre communication (par messages) de contraintes et communication de canaux, nous définissons deux primitives de communication de type *send* et deux de type *receive*.

6.1 Langage et sémantique

Considérons une partition de l'ensemble des variables en deux sous-ensembles non vides : $V = Vca \cup Vco$.
Les éléments de Vca (resp. Vco) sont appelés *variables de canaux* (resp. *variables de contraintes*).

Nous définissons un langage CC adapté à la programmation distribuée que nous appelons DCC[1] (CC distribués). Ce langage est un sous-langage des MCC. Nous y intégrons les agents de type *lien* ainsi que la communication de liens définis à la section 5.5.

Syntaxe :

$$
\begin{array}{lcll}
P & ::= & Dec.A & \text{[prog. et agent initial]} \\
Dec & ::= & \epsilon \mid p(\vec{u};\vec{x}) :: A \mid Dec.Dec & \text{[déclarations]} \\
A & ::= & c \mid ask(d).A \mid u = v \mid ask(u = v) & \\
 & & \mid send(u, \lambda\vec{y}c) \mid receive(u,\vec{y}).A & \text{[agents]} \\
 & & \mid send_{ca}(u,\vec{v}) \mid receive_{ca}(u,\vec{v}).A & \\
 & & \mid lien((\theta,\vec{x}),\vec{x}') \mid send(u,(\theta,\vec{x})) & \\
 & & \mid A \parallel A \mid A + A \mid p(\vec{u};\vec{x}) \mid \exists_x A \mid \exists_u A &
\end{array}
$$

x est une variable de contrainte. u et v sont des variables de canaux. $\vec{x}$ et $\vec{x}'$ sont des vecteurs de variables de contraintes. $\vec{y}$ est un vecteur linéaire de variables de contraintes. $\vec{u}$ est un vecteur de variables de canaux. $\vec{v}$ est un vecteur linéaire de variables de canaux. $\lambda\vec{y}c$ est une abstraction de contrainte vérifiant $\vec{y} \subseteq vc(c)$. θ est un opérateur de store. c est une contrainte vérifiant $vc(c) \cap Vca = \emptyset$. d est un contrainte vérifiant la condition 3 décrite page 81, à savoir : supposons $e \sqcup e' \vdash d$ avec $e \not\vdash d$ alors $vc(e') \neq \emptyset$. p est un symbole de procédure appartenant à un ensemble dénombrable de symboles de procédures auquel est associé un fonction d'arité. Cette fonction fait correspondre à chaque symbole de procédure un couple d'entiers.

Toute déclaration $p(\vec{u};\vec{x}) :: A$ vérifie : A est clos sur ses variables de contraintes qui ne font pas partie de $\vec{x}$. Le but B d'un programme $D.B$ est lui aussi clos sur ses variables de contraintes. Cela revient à restreindre l'utilisation de variables globales : celles-ci ne peuvent être que des variables de canaux.

Remarque : Etant donné une abstraction de contrainte $\lambda\vec{x}c$, nous avons $vc(c) \subseteq \vec{x}$ implique $\vec{x} = vc(c)$ (car $\vec{x} \subseteq vc(c)$ par définition 5.1).

Etant donné un système de contraintes $\mathcal{C}$ et un ensemble de symboles de procédures Π, nous notons $Agents^{\mathcal{C},\Pi}$ l'ensemble des agents définis par cette syntaxe. Nous noterons celui-ci *Agents* lorsque cela ne génère pas d'ambiguïté.

Nous appellerons *agents DCC* les agents définis par cette syntaxe lorsqu'il

[1] Distributed Concurrent Constraint programming

y a risque de confusion avec les agents CC ou les agents MCC.

Les nouveaux agents introduits ici (par rapport aux MCC et à l'exception des agents *lien*) ne sont que des sucres syntaxiques destinés à assurer la séparation entre variables de canaux et variables de contraintes. En particulier, $send_{ca}(u, v)$ est un sucre syntaxique pour $\exists_x send(u, \lambda xx = v)$ et $p(\vec{u}; \vec{x})$ peut être remplacé par $p(\vec{y})$ où $\vec{y}$ est le vecteur composé des variables de $\vec{u}$ et de celles de $\vec{x}$. D'autre part, la présence de deux opérateurs de restriction ne sert qu'à mieux expliciter la séparation des variables. Cela reviendrait au même que de définir un opérateur $\exists_t$ avec $t \in V$.

Un programme écrit dans ce langage est donc un programme MCC. Sa sémantique est celle définie précédemment.

Afin de bien clarifier le mécanisme de communication de canaux, nous introduisons la règle de réduction suivante (notons qu'il sagit en fait d'un cas particulier de la règle (communication)).

$$\frac{d \vdash u = u' \qquad d \neq faux \qquad long(\vec{v}) = long(\vec{v}')}{d \parallel send_{ca}(u, \vec{v}) \parallel receive_{ca}(u', \vec{v}').A \rightarrow d \parallel \vec{v} = \vec{v}' \parallel A} \qquad (\text{communication}_{ca})$$

Pour une meilleure compréhension, nous pouvons aussi remplacer la règle (récursion) des DCC par la règle suivante.

$$\frac{p(\vec{u}; \vec{x}) :: A \in D \qquad \vec{u} \cap \vec{v} = \emptyset \qquad \vec{x} \cap \vec{y} = \emptyset}{p(\vec{v}; \vec{y}) \rightarrow \exists_{\vec{u}} \exists_{\vec{x}} (\vec{u} = \vec{v} \parallel \vec{x} = \vec{y} \parallel A)} \qquad (\text{récursion})$$

La congruence structurelle est celle de la sous-section 3.2.2 définie sur les agents DCC.

Les notion d'agent en forme standard, de store et d'observable (store observable) sont héritées de la section 5.3. Seule est (très légèrement) modifiée la définition de la forme standard pour prendre en compte les agents *lien* :

Définition 6.1 Un agent A est en *forme standard* s'il est de la forme :
$A = \exists_{x1} ... \exists_{xk} (d \parallel A_1 \parallel ... \parallel A_n)$
où d est une contrainte et les A_i sont :

soit des agents simples ask, appels de procédure, agents de communication ou liens (i.e. ce ne sont pas des contraintes),
soit des agents ΣB_i. $\square$

Etant donné un programme DCC, les règles (parallel), (exists), (équiv), (choix) de la sous-section 3.2.3, les règles de réduction (ask), (communication) de la sous-section 5.3.1, les règles (propagation$_{lien}$), (communication$_{lien}$) de la section 5.5 et (communication$_{ca}$), (récursion) ci-dessus définissent un système de transitions.

Etant donné un programme DCC P, tout agent pour P est "potentiellement distribuable" dans le sens où, aucun lien n'étant crée entre noms et variables de contraintes, la communication par message ne dépend pas des stores locaux. Un tel agent est alors distribuable par rapport à P si il est la composition parallèle d'agents dont les ensembles de variables de contraintes sont disjoints.

Nous étendons aux agents DCC la notion de réduction initiale du chapitre 3 (Def. 3.15).

Définition 6.2 Soit $P \rightarrow P'$ une transition de T, soit $S \rightarrow S'$ l'unique application d'une règle de réduction qui intervient dans $P \rightarrow P'$.
$S \rightarrow S'$ est appelée *réduction initiale* (de la transition).
Si $S \rightarrow S'$ est de type (récursion) ou (propagation$-lien$) nous appelons *occurence initiale* l'occurence de S dans P.
Si $S = d \parallel ask(c).A \rightarrow S'$ est de type (ask) alors l'occurence initiale est celle de $ask(c).A$ dans P.
Si $S \rightarrow S'$ est de type communication (de contrainte, de canal, ou de lien), avec $S = d \parallel send* \parallel receive*$ ($send*$ (resp. $receive*$) est un agents de type $send$ ou $send_{ca}$ (resp. $receive$ ou $receive_{ca}$)), alors l'occurence initiale est celle de $send* \parallel receive*$ dans P. $\square$

La notion de distribuabilité s'étend de manière évidente à la communication de canaux et à la communication de liens. Nous nous contentons de donner ici les modifications nécessaires à la définition 5.5. (les définitions 5.6 et 5.7 n'ont pas besoin d'être modifiées et les propositions de la section 5.4 restent valides).

Définition 6.3 Un agent A pour P est *distribuable en A' et A"* (par rapport à P) s'il est structurellement équivalent à un agent de la forme
$\exists_{\vec{x}}(A' \parallel A'')$
et si pour toute réduction $A \rightarrow B$ on a :

- Soit $A \rightarrow B$ est de type (propagation$_{lien}$), $B \equiv \exists_{\vec{x}}(d \parallel A' \parallel A'')$ pour une certaine contrainte d et soit B est distribuable en $(d \parallel A')$ et A'', soit B est distribuable en A' et $(d \parallel A'')$.
- Soit il existe B' tel que $A' \rightarrow B'$ et B est distribuable en B' et A''.
- Soit il existe B'' tel que $A'' \rightarrow B''$ et B est distribuable en A' et B''.
- Soit $A' \equiv (send(u, \lambda\vec{y}c) \parallel \tilde{A}')$, $A'' \equiv (receive(v, \vec{z}).\hat{A}'' \parallel \tilde{A}'')$, A se réduit en $\exists_{\vec{x}}(\tilde{A}' \parallel c[\vec{z}/\vec{y}] \parallel \hat{A}'' \parallel \tilde{A}'')$, et B est distribuable en $\tilde{A}'$ et $(c[\vec{z}/\vec{y}] \parallel \hat{A}'' \parallel \tilde{A}'')$.
- Soit $A'' \equiv (send(u, \lambda\vec{y}c) \parallel \tilde{A}'')$, $A' \equiv (receive(v, \vec{z}).\hat{A}' \parallel \tilde{A}')$, A se réduit en $\exists_{\vec{x}}(c[\vec{z}/\vec{y}] \parallel \hat{A}' \parallel \tilde{A}' \parallel \tilde{A}'')$ et B est distribuable en $(c[\vec{z}/\vec{y}] \parallel \hat{A}' \parallel \tilde{A}')$ et $\tilde{A}''$.
- Soit $A' \equiv (send_{ca}(u, u') \parallel \tilde{A}')$, $A'' \equiv (receive_{ca}(v, v').\hat{A}'' \parallel \tilde{A}'')$, A se réduit en $\exists_{\vec{x}}(\tilde{A}' \parallel u' = v' \parallel \hat{A}'' \parallel \tilde{A}'')$, et B est distribuable en C et $(u' = v' \parallel \hat{A}'' \parallel \tilde{A}'')$, avec $(C \parallel u' = v' \parallel \hat{A}'' \parallel \tilde{A}'') \equiv \exists_{\vec{x}}(\tilde{A}' \parallel u' = v' \parallel \hat{A}'' \parallel \tilde{A}'')$.
- Soit $A'' \equiv (send_{ca}(u, u') \parallel \tilde{A}'')$, $A' \equiv (receive_{ca}(v, v').\hat{A}'' \parallel \tilde{A}')$, A se réduit en $\exists_{\vec{x}}(u' = v' \parallel \hat{A}' \parallel \tilde{A}' \parallel \tilde{A}'')$, et B est distribuable en $(u' = v' \parallel \hat{A}' \parallel \tilde{A}')$ et C, avec $(C \parallel u' = v' \parallel \hat{A}' \parallel \tilde{A}') \equiv \exists_{\vec{x}}(u' = v' \parallel \hat{A}' \parallel \tilde{A}' \parallel \tilde{A}''$.
- Soit $A' \equiv (send(u, (\theta, \vec{y})) \parallel \tilde{A}')$, $A'' \equiv (receive(v, \vec{z}).\hat{A}'' \parallel \tilde{A}'')$, A se réduit en $\exists_{\vec{x}}(\tilde{A}' \parallel lien((\theta, \vec{y}), \vec{z}) \parallel \hat{A}'' \parallel \tilde{A}'')$, et B est distribuable en $\tilde{A}'$ et $\exists_{\vec{x}}(\tilde{A}' \parallel lien((\theta, \vec{y}), \vec{z}) \parallel \hat{A}'' \parallel \tilde{A}'')$.
- Soit $A'' \equiv (send(u, (\theta, \vec{y})) \parallel \tilde{A}'')$, $A' \equiv (receive(v, \vec{z}).\hat{A}'' \parallel \tilde{A}')$, A se réduit en $\exists_{\vec{x}}(lien((\theta, \vec{y}), \vec{z}) \parallel \hat{A}' \parallel \tilde{A}' \parallel \tilde{A}'')$, et B est distribuable en $\exists_{\vec{x}}(lien((\theta, \vec{y}), \vec{z}) \parallel \hat{A}' \parallel \tilde{A}' \parallel \tilde{A}''$ et $\tilde{A}''$.

□

Le traitement de la communication de canaux fait intervenir un agent intermédiaire (l'agent C dans la définition) car la contrainte $u' = v'$ peut, même dans le cas où elle ne porte que sur des canaux locaux, apporter suffisament

d'information au store local pour que celui-ci ait pour conséquence l'égalité de deux canaux globaux. Il faut alors que cette information soit "transmise" à l'autre store local. Pour plus de précision concernant ce mécanisme, se reporter au lemme 6.3 ci-dessous.
Notons que ce phénomène est ici mis en évidence par l'utilisation d'agents de communication spécifiques pour la communication de canaux. Il se retrouve cependant dans les MCC et la notion de distribuabilité de la définition 5.5 est moins forte que celle définie ci-dessus. Certains agents MCC sont non distribuables pour la définition 5.5 car celle-ci ne tient pas compte de la propagation d'égalités entre variables de canaux globales qui peut résulter de la communication par messages.

Le traitement des agents *lien* ne pose pas de problème particulier.

La notion de *site* définie ci-dessous caratérise une classe d'agents qui, ne peuvent communiquer avec un éventuel environnement que par messages.

Définition 6.4 Soit A un agent.
A est un *site* s'il est clos sur ses variables de contraintes. □

Nous n'avons pas choisi ce nom de site par hasard. La proposition suivante montre que deux sites en composition parallèle sont distribuables.

Remarquons cependant que cette définition est relativement restrictive quand à l'utilisation d'agents *lien*. Considérons un agent A possédant un sous-agent $lien((\theta, \vec{y}), \vec{z})$. Si les variables de $\vec{y}$ (ou certaines d'entre elles, afin de simplifier nous supposerons qu'il sagit de toutes les variables de $\vec{y}$) n'apparaîssent dans aucun autre sous-agent de A (sauf peut-être dans le premier argument de sous-agents *lien*) alors aucune contrainte ne peut être posée entre ces variables et d'autre variables de contraintes dans A. En supposant que A soit clos sur ses variables de contraintes sauf sur celles de $\vec{y}$. (ce cas est important puisqu'il correspond, en particulier, à la transmission du résultat d'un calcul local), il faudrait alors considérer que celles-ci ne font pas partie de A. L'agent A n'est pas un site. Il vérifie néanmoins la propriété souhaitée, à savoir qu'il ne peut communiquer avec l'environnement que par messages. Il conviendrait donc d'étendre la notion de site. Notons qu'il serait aisé de définir des conditions syntaxiques (par exemple avec un typage dans le style de celui que nous avons utilisé précédemment) dans le but obtenir une notion de site moins restrictive que celle de la définition 6.4 et pour laquelle la proposition ci-dessous soit toujours valide.

Proposition 6.1 Soit P un programme DCC. Soient A_1 et A_2 deux agents pour P vérifiant : A_1 et A_2 sont des sites.
$A = A_1 \parallel A_2$ est distribuable en A_1 et A_2 par rapport à P. $\Box$

Lemme 6.1 Soit A un site.
Tout agent B vérifiant $B \equiv A$ est un site. $\Box$

preuve :
Immédiat, par induction.
$\Box$

Lemme 6.2 Soit $P = D.B$ un programme DCC. Soit A un site.
Si $A \rightarrow A'$ est une transition du système de transitions associé à P alors A' est un site. $\Box$

preuve :
Soit $S \rightarrow S'$ la transition initiale de $A \rightarrow A'$.
Soit B vérifiant $B \equiv A$, $B[S] \rightarrow B[S'/S]$ et $B[S'/S] \equiv A'$.
B est un site (lemme 6.1).
Montrons que $B[S'/S]$ est un site.
Si $S \rightarrow S'$ est une réduction de type :
1. (ask), (communication), (propagation$_{lien}$), (communication$_{lien}$) ou (communication$_{ca}$) alors il est évident que l'ensemble des variables de contraintes libres de S' est inclus dans l'ensemble des variables de contraintes libres de S.
Donc, $B[S'/S]$ est clos sur ses variables de contraintes puisque B l'est.
2. (récursion) , avec $S = p(\vec{u}; \vec{x})$ alors
Soit $p(\vec{v}; \vec{y}) :: C$ la déclaration de p dans D.
Par hypothèse, C ne possède aucune variable de contrainte libre, donc $S' = \exists_{\vec{v}} \exists_{\vec{y}} (\vec{v} = \vec{u} \parallel \vec{y} = \vec{x} \parallel C)$ non plus, et $B[S'/S]$ est un site.

Nous avons montré que $B[S'/S]$ est un site.
D'autre part, A' est structurellement équivalent à $B[S'/S]$
Finalement, A' est un site (par lemme 6.1).
$\Box$

Le lemme suivant met en évidence le mécanisme de “propagation” des contraintes sur les canaux. Si le store d'un site à pour conséquence l'égalité de deux variables de canaux globales (ou du moins non locales à ce site), cette égalité est "transmise" aux autres sites (qui ont accès à ces variables) par le simple jeu de l'axiome $c \parallel d \equiv c \sqcup d$ de la congruence structurelle. Nous avons écrit ci-dessus "propagation" et "transmise" entre guillemets car ce mécanisme est traité, au niveau théorique, par la congruence structurelle. Il n'y a réellement propagation qu'au niveau de l'implantation. Il est alors nécessaire de diffuser cette information à tous les sites pour lesquels les opérateurs de restriction donnnent accès à ces variables. Notons que cette opération de diffusion est effectuée sur des cas simples de contraintes.

Lemme 6.3 Soient A_1 et A_2 deux sites. Soient u et v deux variables de canaux vérifiant : u (resp. v) est soit globale à A_1 et A_2 soit locale à A_1.
Si $store(A_1 \parallel A_2) \vdash u = v$ alors il existe A'_1 vérifiant :
$(A_1 \parallel A_2) \equiv (A'_1 \parallel A_2)$ et $store(A'_1) \vdash u = v$. □

preuve :
Aucune des deux variables n'étant locale à A_2, les contraintes dans A_2 ne peuvent intervenir que dans la mesure ou $store(A_2)$ a pour conséquence l'égalité de deux variables globales.
Supposons $store(A_2) \vdash w = w'$, w et w' étant deux variables de canaux globales.
Nous avons $A_2 \equiv (A_2 \parallel w = w')$.
Donc $(A_1 \parallel A_2) \equiv (A_1 \parallel w = w' \parallel A_2)$ et $store(A_1 \parallel w = w') \vdash w = w'$.
□

preuve de la proposition 6.1 :
Il nous faut montrer que toute transition $A \to A'$ vérifie les propriétés de la définition 6.3 et que les agents en composition parallèle qui forment A' (selon la définition 6.3) sont des sites.
Si $A \to A'$ est de type :
1. (récursion) alors, elle ne fait intervenir dans sa réduction initiale qu'un unique agent simple. Celui-ci est donc soit dans A_1 soit dans A_2.
Supposons le dans A_1, nous avons $A' \equiv A'_1 \parallel A_2$ avec $A_1 \to A'_1$.
De plus, par lemme 6.2, A'_1 est un site.
2. (ask) alors, soit $d \parallel ask(c).D \to d \parallel D$ sa réduction initiale.
Si $c = vrai$ c'est trivial.
Sinon, nous avons $vc(c) \neq \emptyset$ (par hypothèses sur les agent ask dans les DCC, de plus $c \neq faux$ à cause des prémisses de la règle (ask)).
Soit $x \in vc(c)$. Nous avons $x \in vc(d)$ (car $d \vdash c$ et par hypothèses sur le système

de contraintes).
De plus, A_1 et A_2 sont clos sur leurs variables de contraintes (ce sont des sites)
Donc il existe d' tel que $d' \vdash c$ et d' et $ask(c)$ appartiennent soit tous deux à A_1 soit tous deux à A_2.
On conclut comme en 1.
3. (communication) alors, soit $S \to S'$ sa réduction initiale, avec
$S = d \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).B$ et $S' = d \parallel c[\vec{y}/\vec{x}] \parallel B$.
Si $send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).A$ est une occurence de A_1 (on effectue le même raisonnement pour A_2) alors il existe une occurence d' de A_1 telle que $d' \vdash u = v$ (lemme 6.3).
Donc $d' \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).B$ est une occurence de A_1 et on conclut comme en 1.
Supposons que $send(u, \lambda\vec{x}c)$ (resp. $receive(v, \vec{y}).B$) soit une occurence de A_1 (resp. A_2) (on effectue le même raisonnement mutatis mutandis). Celles-ci sont en composition parallèle de premier niveau.
Il existe $\tilde{A}_1$ et $\tilde{A}_2$ tels que $A_1 = \exists_{\tilde{z}_1}(send(u, \lambda\vec{x}c) \parallel \tilde{A}_1)$ et,
$A_2 = \exists_{\tilde{z}_2}(receive(v, \vec{y}).B \parallel \tilde{A}_2)$.
$\tilde{A}_1$ est (trivialement) un site.
D'autre part, $\vec{x} = vc(c)$, donc $cv(c[\vec{y}/\vec{x}]) = \vec{y}$. D'où $\tilde{A}_2 \parallel c[\vec{y}/\vec{x}]$ est un site.
4. (communication$_{ca}$) et (communication$_{lien}$) : la preuve suit exactement même schéma qu'en 3.
5. (propagation*lien*) alors soit
$d \parallel lien((\theta, \vec{x}), \vec{y}) \to d \parallel lien((\theta, \vec{x}), \vec{y}) \parallel \theta(\vec{x}, d)[\vec{y}/\vec{x}]$ sa réduction iniiale.
Supposons que l'agent *lien* soit dans A_1. Puisque A_1 est un site, $\vec{y}$ est local à A_1. d'où $A_1 \parallel \theta(\vec{x}, d)[\vec{y}/\vec{x}]$ est un site. □

Corollaire 6.1 Soit P un programme DCC. Un agent $A = \exists_{\vec{x}}(A_1 \parallel ... \parallel A_n)$ pour P dans lequel les A_i sont des sites est distribuable en $A_1, ..., A_n$ par rapport à P. □

preuve :
Immédiat par les propositions 6.1, 5.7 et 5.8. □

6.2 Exemples

Les exemples que nous décrivons dans cette section mettent en évidence le pouvoir d'expression des langages DCC. Nous modélisons, entre autres, des problèmes classiques de la programmation distribuée. La notion de consommation de ressource introduite par la communication par messages se révèle d'une grande utilité pour la conception de programmes réactifs ou interactifs. Il sagit là d'une caractéristique qui fait grandement défaut dans les langages CC (cf. [BGJK96a] pour une extension des CC intégrant une notion de consommation de ressource). En particulier, elle permet la simulation de variables impératives d'une manière bien plus élégante que celle employée habituellement (liste historique des valeurs prises par la variable) dans les LLC, les CC et la PL en général.

Ces exemples utilisent le système de contraintes de Herbrand défini au chapitre 2.

Nous utiliserons le sucre syntaxique suivant : Si a est une constante du système de contraintes alors $send(u, a)$ dénote $\exists_x send(u, \lambda xx = a)$.
Les lettres a, b, c, d, e (resp. x, y, z) sont des méta-variables de variables de canaux (resp. de contraintes). Nous ommettrons parfois la flèche sur les vecteurs de variables écrits en extension.

6.2.1 Reconfiguration

La reconfiguration dynamique du réseau de communication revêt une grande importance pour la concurrence et la distribution. Cela s'effectue dans les CC par l'intermédiaire de variables impératives modélisées par des listes (cf. [LM92] et notre exemple 3.4). Dans les MCC, elle est obtenue d'une façon beaucoup plus naturelle par communication de noms (variables de canaux) à la manière du π-calcul.

Dans l'exemple suivant, deux agents p et q partagent un nom a et p partage un nom b avec un troisième agent r.

Considérons le programme dont les déclarations de procédures sont (les déclarations de s et t ne sont pas spécifiées) :

$$\begin{array}{lll} p(a,b) & :: & send_{ca}(b,a) \parallel \exists_{\vec{y}} receive(a,\vec{y}).s(\vec{y}) \\ q(a) & :: & send(a, \lambda\vec{x}c) \\ r(b) & :: & \exists_d(receive_{ca}(b,d).\exists_{\vec{y}} receive(d,\vec{y}).t(\vec{y})) \end{array}$$

et le but : $\exists_{a,b}(p(a,b) \;\parallel\; q(a) \;\parallel\; r(b))$

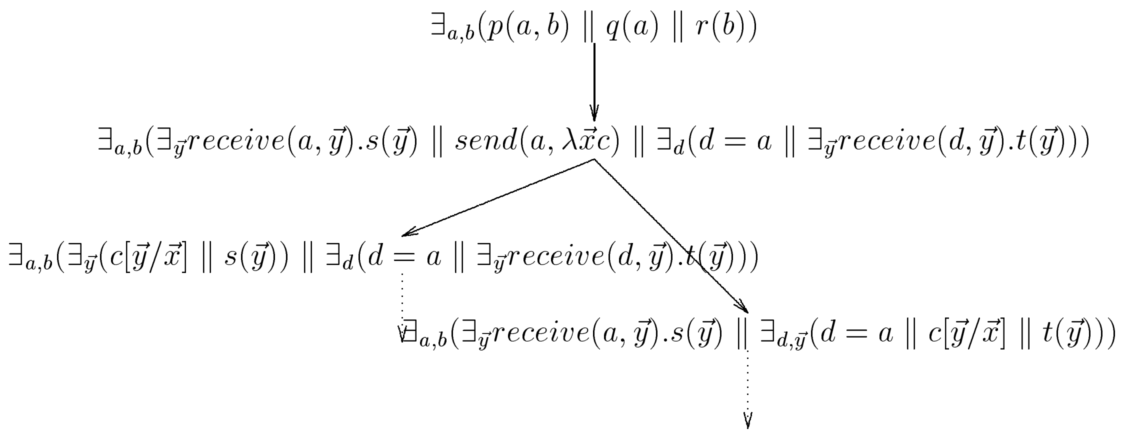

Figure 6.1 : Système de transitions pour l'exemple 6.2.1

Après communication du nom a sur le nom b, a est partagé par les trois sites. $send(a, \lambda\vec{x}c)$ peut alors communiquer avec l'un ou l'autre des deux autres sites.
Il y a communication d'un nom de canal, plus précisément cela correspond à la "scope extrusion" du π-calcul. La figure 6.1 représente le système de transitions (sur les classes d'équivalence d'agents) associé à ce programme.

6.2.2 Client-Serveur

De nombreux problèmes de programmation nécessitent une certaine non monotonie de l'information manipulée. Ceux-ci sont souvent difficiles à résoudre en CC à cause de la monotonie du store.

En particulier, les programmes de type client-serveur sont souvent écrits en utilisant des listes pour implanter des variables impératives (i.e. non monotones). Nous proposons ci-dessous une version plus naturelle de ce problème.

$$\begin{array}{lll} client(a) & :: & \exists_b(send_{ca}(a,b) \parallel \exists_{\vec{y}}receive(b,\vec{y}).action(\vec{y})) \\ serveur(a) & :: & \exists_b(receive_{ca}(a,b).\exists_{\vec{y}}send(b,\lambda\vec{y}c) \parallel serveur(a)) \end{array}$$

Un but type consistera en un certain nombre de clients et de serveurs partageant un même paramètre a et s'exécutant en parallèle. a est le canal sur lequel les clients envoient leurs requêtes. La confidentialité est assurée par la définition, au début de la transaction, d'un nouveau canal (b) qui est envoyé au serveur. Ce canal est donc un canal privé, connu seulement du client et du serveur. La réponse sera communiquée par son intermédiaire.

Un agent $client(a)$ ou $serveur(a)$ est bien sûr un site au sens de la définition 6.4.

Le programme précédent ne modélise en fait que le protocole de communication. Le client ne pose pas de requête et le serveur n'effectue pas de calcul avant d'envoyer sa réponse.

Nous donnons ci-dessous une version plus effective. Cet exemple montre le mécanisme de calcul local en DCC ainsi que celui de transmission du résultat.

$$\begin{array}{lll} client(a) & :: & \exists_{b,c}(send_{ca}(a, \overrightarrow{(b,c)}) \parallel \exists_{\vec{y}} send(b, \lambda\vec{y}request) \\ & & \parallel \exists_{\vec{z}} receive(c, \vec{z}).action(\vec{z})) \\ serveur(a) & :: & \exists_{b,c}(receive_{ca}(a, \overrightarrow{(b,c)}).\exists_{\vec{y},\vec{z}}(receive(b, \vec{y}) \parallel calcul(\vec{y}, \vec{z}) \\ & & \parallel send(c, (result, \vec{x})) \parallel serveur(a)) \end{array}$$

avec $\vec{x} \subseteq \vec{y} \cup \vec{z}$.

Les canaux b et c sont des canaux privés, locaux au serveur et à son client. b est le canal par l'intermédiaire duquel le client pose sa requête. Le serveur lui envoie sa réponse sur le canal c. a est un canal public qui ne sert qu'à initialiser la transaction.
$\lambda\vec{y}request$ est une abstraction de contrainte qui contient – qui code – la requête du client. la procédure *calcul* effectue le calcul demandé sur un store local formé des variables des vecteurs $\vec{y}$ et $\vec{z}$. Le résultat est envoyé au client par l'intermédiaire d'un opérateur de store ($result$) qui porte sur certaines variables de ce store local.

6.2.3 Le problème du dîner des philosophes

Le problème du dîner des philosophes est classique en concurrence. n philosophes sont assis autour d'une table ronde. Il y a une fourchettte entre chacun d'eux. Chaque philosophe a besoin des deux fourchettes situées sur ses cotés pour manger. Le problème est de réaliser l'accès en exclusion mutuelle des philosophes aux fourchettes.

Nous donnons une version du dîner des philosophes qui combine communication par messages et communication ask/tell. Elle est inspirée d'une version en Linda ([CG89]) en ce qu'elle repose sur l'existence de $n-1$ jetons pour n philosophes. Un philosophe doit posséder un jeton pour prendre les fourchettes. Ce petit "trick" rend l'exclusion mutuelle plus facile à assurer car l'atomicité de la prise des deux fourchettes n'est plus nécessaire. Ce programme illustre aussi la reconfiguration dynamique de réseau de communication, ce dernier n'étant pas fixé par le but – agent initial – mais créé par les procédures *init* et *init*2. Nous utilisons à cet effet un compteur qui suppose, en plus du système de contraintes de Herbrand, un système de contraintes sur les domaines finis.

$$
\begin{aligned}
init(n) :: \exists_{a,t,f}(\ & send(a,1)\\
& \parallel send(f,taken)\\
& \parallel init2(a,t,f,f,n))
\end{aligned}
$$

$$
\begin{aligned}
init2(a,t,f,f',n) :: \exists_{n'} receive(a,n').(\ & \\
& ask(n' < n).\exists_{f'',n''}(send(f'',taken)\\
& \qquad \parallel send(t)\\
& \qquad \parallel phil(f',f'',t,n')\\
& \qquad \parallel n'' = n'+1\\
& \qquad \parallel send(a,(proj,n''))\\
& \qquad \parallel init2(a,t,f,f'',n))\\
& \parallel ask(n' = n).phil(f',f,t,n))
\end{aligned}
$$

$$
\begin{aligned}
phil(f,f',t,n) :: \exists_{x,x'}\ (receive(t).(\ & receive(f,x).true\\
& \parallel receive(f',x').true\\
& \parallel ask(x = taken \wedge x' = taken).(\\
& \qquad send(f,taken)\\
& \qquad \parallel send(f',taken)\\
& \qquad \parallel send(t)\\
& \qquad \parallel phil(f,f',t,n))))
\end{aligned}
$$

Le but est : $\exists_n(n = N \parallel init(n))$

Les fourchettes sont modélisées par des canaux. La disponibilité d'une fourchette f est symbolisée par la présence d'un agent $send(f, taken)$, où $taken$ est une constante particulière du système de contraintes. Cette façon de procéder est très proche de celle utilisée dans la version en Pict ([Pie96]) du problème du dîner des philosophes dont nous nous sommes d'ailleurs inspiré. L'action de prendre la fourchette f consiste en la consommation de l'agent $send(f, taken)$ (réception sur f). Le philosophe libèrera la fourchette en introduisant un nouvel agent $send(f, taken)$.

Les tickets sont modélisés par un canal t sur lequel sont initialisés $n-1$ agents $send(t)$. Ceux-ci étant utilisés dans un but de synchronisation pure, l'abstraction de contrainte transmise est sans importance. $send(t)$ (resp. $receive(t)$) est un sucre syntaxique pour $send(t, const)$ (resp. $\exists_x receive(t, x)$), $const$ étant une constante quelconque du système de contraintes de Herbrand. L'action de prendre un ticket consiste en la consommation d'un agent $send(t)$.

Les procédures $init$ et $init2$ servent à initialiser les agents $phil$ ainsi que le réseau de communication. $init$ initialise la première fourchette. $init2(a, t, f, f', n)$ initialise une fourchette f'' et le philosophe numéro n (n est une variable de contrainte, en toute rigueur nous devrions écrire $init((a, \overrightarrow{t, f}, f'), n)$ car $a, \overrightarrow{t, f}, f'$ est un vecteur de variables de canaux.) sur les fourchettes f' et f''. Remarquons que nous avons choisi de munir chaque philosophe d'un identificateur unique (son numéro d'ordre d'initialisation). Cet identificateur est cependant inutile, on voit d'ailleurs que n n'intervient nulle part dans $phil(f, f', t, n)$. Notons l'utilisation d'un opérateur de projection sur le canal a pour gérer – incrémenter – le compteur n qui représente dans $init2$ le nombre de philosophes déjà initialisés. La fourchette f est la première fourchette initialisée par $init$. Elle servira à "fermer le cercle" lors de l'initialisation du dernier philosophe. Ce mécanisme d'initialisation utilise la reconfiguration pour créer dynamiquement le réseau.

Après l'initialisation, nous obtenons l'agent suivant (pour N=2):

$$
\begin{aligned}
\exists_n \exists_{t, f_1, f_2}(\ & n = 2 \parallel send(t) \parallel send(f_1, taken) \parallel send(f_2, taken) \\
& \parallel \exists_{n_1}(n_1 = 1 \parallel phil(f_1, f_2, t, n_1)) \\
& \parallel \exists_{n_2}(n_2 = 2 \parallel phil(f_2, f_1, t, n_2)))
\end{aligned}
$$

Les agents $\exists_{n_1}(n_1 = 1 \parallel phil(f_1, f_2, t, n_1))$ et $\exists_{n_2}(n_2 = 2 \parallel phil(f_2, f_1, t, n_2))$ sont des sites.

6.2.4 L'algorithme d'élection de Chang et Roberts

Il sagit d'un algorithme classique d'élection sur un anneau unidirectionnel. On suppose qu'à chaque site est associé un identificateur unique et que l'on dispose d'un ordre total sur les identificateurs. Le problème consiste à trouver le site d'identificateur maximal et à annoncer son identité (i.e. son identificateur) à tous les sites.

Les sites ne connaissent aucune information structurelle concernant le réseau de communication, ni l'ordre N du réseau, ni l'identité de leurs voisins. Les sites savent seulement distinguer localement leurs canaux d'entrée (site prédécesseur) et de sortie (site successeur).

Afin de simplifier cet exemple, nous ne modélisons pas l'initialisation des sites et du réseau, contrairement à l'exemple précédent. Le but consiste donc en un ensemble de sites en composition parallèle, chacun envoyant son identificateur à son successeur. Lorqu'un site reçois un message, trois situations peuvent se produire.

- Si son identificateur propre est supérieur à celui reçu alors le message est "éteint" : le site n'effectue aucune action et se met en attente d'un autre message.
- S'il est supérieur alors le message est transmis à l'identique et le site se met en attente d'un autre message.
- S'il est égal alors le site a gagné l'élection et envoie un message annoncant qu'il est l'élu.

$$
\begin{array}{lll}
CR(a,b,id) :: \exists_{y,winner}(& receive(a,y).(& \\
& \parallel ask(y < id).CR(a,b,id) & \\
& \parallel ask(y > id).(send(b,(proj,y)) \parallel CR(a,b,id)) & \\
& \parallel ask(y = id).(\ winner = id & \\
& \qquad\qquad \parallel \exists_x\ (x = won(id) \parallel send(b,(proj,x))) & \\
& \qquad\qquad \parallel \exists_x receive(a,x).vrai)) & \\
& \parallel ask(\exists_x y = won(x)).\exists_x(\ y = won(x) & \\
& \qquad\qquad\qquad \parallel winner = x & \\
& \qquad\qquad\qquad \parallel send(b,(proj,y)))) &
\end{array}
$$

Le but est :

$$\begin{array}{ll}\exists_{a_1 \ldots a_N}(& \exists_{id_1}(CR(a_1, a_2, id_1) \parallel id_1 = Id_1 \parallel send(a_2, Id_1)) \\ & \parallel \ \ldots \\ & \parallel \ \exists_{id_N}(CR(a_N, a_1, id_N) \parallel id_N = Id_N \parallel send(a_1, Id_N)))\end{array}$$

Ce programme ne nécessite pas de commentaire particulier sauf en ce qui concerne la terminaison de l'élection. Lorsque un site est élu (cas $y = id$), il pose la contrainte $winner = id$ (il “affecte” sa variable locale $winner$ à la valeur de son identificateur) puis il envoie un message à son successeur contenant $won(Id)$, won étant un symbole de fonction du système de contraintes de Herbrand. Nous supposons ici la possibilité de construire un terme avec un symbole de fonction et une constante d'un système de contraintes sur les domaines finis, comme cela se fait dans les implantations des langages de CLP. Ce message parcourera l'ensemble du réseau puis sera “éteint” par l'agent $\exists_x receive(a, x).vrai$ du site élu.

Pour qu'un site non élu reconnaisse le message annonçant le résultat de l'élection, nous avons alors besoin d'utiliser un opérateur ask sur la contrainte $\exists_x y = won(x)$, qui fait appel à un opérateur de cylindrification – Il est en effet clair qu'un agent $\exists_x ask(y = won(x))$ ne serait pas débloqué en présence de la contrainte $y = won(Id)$, Id étant l'identificateur du site élu. Pour s'en persuader, il suffit de se rendre compte que $(\exists_x ask(y = won(x)) \parallel y = won(Id))$ est strucurellement équivalent à $\exists_x(ask(y = won(x)) \parallel y = won(Id))$. Cela est tout à fait valide dans la sémantique des MCC mais il ne semble pas habituel, du moins à notre connaissance, de permettre l'utilisation explicite d'opérateurs de cylindrification dans le langage, le programmeur étant limité à l'utilisation d'opérateurs de restriction, qui ne se placent pas au même niveau. Ce problème – qui n'en est pas vraiment un – est hérité des CC dans lesquels il se retrouve. Il provient du fait que, replacé dans le contexte du système de contraintes de Herbrand, l'opérateur ask correspond au filtrage et non pas à l'unification (cf. l'exemple 3.2 du chapitre 3 qui montre un cas similaire dans lequel un $tell$ est nécessaire après le ask afin d'effectuer l'unification). Le mécanisme de synchronisation des CC nécessite dans certains cas l'utilisation d'opérateurs de cylindrification dans les agents ask.

Ce qui par contre est un vrai problème, est que la granularité très fine des CC demanderait, pour bien des applications, la mise à disposition d'un opérateur de séquentialité ou du moins, d'une notion étendue d'atomicité. L'exemple du dîner des philosophes ci-dessus est à ce sujet significatif. Nous n'avons modélisé que le protocole de communication. Ajouter dans les agents *phil* un action *eat* semble difficile car il faudrait alors pouvoir décider de la terminaison de cette action avant de libérer le ticket et les fourchettes. Pour une étude de ce problème dans le cadre des langages CC, le lecteur intéressé peut se reporter à [Cha97] dans lequel un protocole d'accès en exclusion mutuelle à une ressource est décrit. Il y est constaté la nécessité d'un mécanisme supplémentaire de synchronisation pour pouvoir exécuter une "véritable" action à l'intérieur de la section critique.

Les MCC apportent une réponse – hélas partielle – à ce problème. La transmission de canaux locaux permet la création d'un canal privé par l'intermédiaire duquel une action peut être exécutée sans interférence avec l'environnement. En résumé les MCC permettent l'accès privé à un site – comme cela est montré par l'exemple *client/serveur* (deuxième version) – mais pas l'accès en exclusion mutuelle à une variable (ou du moins pas avec une "vraie" section critique). Le premier point est hérité du π-calcul et le second des CC. Il conviendrais d'étudier plus précisément quel pouvoir d'expression de la synchronisation confère aux MCC son mécanisme de communication. Dans l'exemple du dîner des philosophes, il semble cependant difficile de construire une "vraie" section critique sans l'aide d'une notion supplémentaire de synchronisation.

Chapitre 7

Observer la capacité de communication

La sémantique opérationnelle développée dans le chapitre précédent associe un ensemble d'observables à un programme mais ne permet pas de caractériser quel serait le comportement de celui-ci par rapport à un environnement (communication avec d'autres programmes par l'intermédiaire de variables libres). Nous définissons dans cette section un système de transitions annoté, à la manière de ce qui est fait pour les CCS et le π-calcul, qui rend compte de la capacité de communication d'un progamme puis nous définissons une relation de bisimilarité. Nous suivons en cela l'idée de Robin Milner selon laquelle observer un processus signifie communiquer avec lui. Un tel travail a été fait par Saraswat et Rinard ([SR90]) pour les langages CC. Mais, bien que nous puissions nous en inspirer, la conjonction de la communication ask/tell et de la communication par messages introduit une complexité nouvelle.

Nous nous restreignons au cas des DCC sans les agents *lien*, et plus précisément au cas des sites DCC (sans agents *lien*). Cette sémantique peut cependant être étendue aux MCC. Nous donnons l'idée de cette extension.

7.1 Une sémantique de bisimulation

La définition d'une telle sémantique pose certains problèmes :

- Observer une transition *ask* signifie observer la quantité d'information manquante dans le store pour que la contrainte argument de l'agent ask soit conséquence de celui-ci. Cela ne peut pas s'effectuer au niveau des agents atomiques puisque le comportement du ask dépend du store, qui est global à un site. D'autre part, caractériser cette "quantité d'information manquante" n'est pas immédiat.

- La *scope extrusion* (cf. exemple 6.2.1) permet l'envoi de canaux locaux. Un mécanisme assurant la cohérence des canaux locaux transmis au long d'une dérivation est donc nécessaire.

- La communication par messages permet à un site d'apporter de l'information concernant des variables locales d'un autre site. Il faut donc, pour ce qui concerne la communication *ask/tell*, observer les échanges d'information non seulement sur les variables de contraintes globales mais aussi sur certaines variables de contraintes locales.

Le premier point se retrouve dans les CC et le second dans le π-calcul. Les DCC sont cependant plus complexes que les CC ou le π-calcul. Le fait d'avoir à traiter conjointement les deux mécanismes de communication a pour conséquence de permettre à l'environnement d'apporter de l'information "complexe" sur des variables locales (troisième point). En effet, dans le cas du π-calcul, cette information est réduite à la substitution de noms de canaux tandis que dans les DCC, celle-ci peut influencer le comportement d'agents *ask* qui pourtant ne portent que sur des variables locales. Dans [SR90], les transitions du système de transitions annoté dépendent d'un ensemble de variables – l'ensemble des variables visibles – qui correspond intuitivement aux variables libres du but qui est réduit. Ces variables sont celles par l'intermédiaire desquelles l'agent considéré peut communiquer avec un éventuel environnement. Dans le cas des DCC, un tel ensemble de variables visibles ne peut être défini que de manière dynamique. Certaines variables, bien que locales, doivent être considérées comme étant visibles.

Comme pour le π-calcul, nous définissons une action *silencieuse* que nous notons τ. Elle correspond au cas d'une transition qui peut s'effectuer indépendamment de l'environnement. Par exemple, l'observable d'une transition (ask-libre) $A[ask(c).B] \rightarrow A'$ est τ dans le cas ou c est conséquence du store. Dans le cas contraire, l'agent *ask* peut être libéré par l'environnement si celui-ci communique l'information nécessaire sur un certain ensemble U de variables (de contraintes) visibles. U est l'ensemble des variables sur lesquelles l'environnement est suceptible d'avoir apporté de l'information. Ce sont donc les variables globales et les variables locales arguments d'un agent *receive* déjà exécuté. Construire U nous impose que cette construction doit être effectuée dynamiquement, et que le nom d'une variable liée doit être "fixé" si elle est ajoutée à U (si une variable liée appartient à U il ne faut pas qu'elle soit modifiée par α-renommage). Supposons cet ensemble U connu, l'agent *ask* peut être réduit si l'environnement apporte une contrainte d vérifiant $vc(d) \subseteq U$, $store(A) \sqcup d \vdash c$ et $store(A) \sqcup d \neq faux$. Afin de capturer une équivalence de capacité de communication, nous devons caractériser l'ensemble des contraintes qui vérifient ces propriétés. Saraswat et Rinard utilisent pour cela un système de contraintes défini sur un langage du premier ordre dans lequel l'implication et la quantification universelle sont des constructeurs sur les contraintes. Soit σ un store, soit c une contrainte, soit U l'ensemble des variables visibles, il existe alors une plus petite contrainte d sur les variables U vérifiant $\sigma \wedge c \neq faux$ et $d \Rightarrow (\sigma \Rightarrow c)$. Soit $\bar{x}$ l'ensemble des variables libres dans σ moins U, on a alors $d = \forall \bar{x}(\sigma \Rightarrow c)$. La structure plus abstraite de système de contraintes que nous utilisons ne nous permet pas de disposer d'outils aussi complexes. En particulier, elle ne nous permet pas la définition d'un opérateur de quantification universelle. De plus, toujours dans le cadre défini au chapitre 2, s'il existe une contrainte minimale vérifiant les propriétés souhaitées, elle n'est en général pas unique.

Il nous est possible de résoudre ce problème en travaillant sur le système de contraintes des parties. Considérons l'ensemble $\mathbf{d}$ des contraintes d vérifiant $vc(d) \subseteq U$ et $store(A) \sqcup d \vdash c$. $\uparrow \mathbf{d}$ est une contrainte du système de contraintes des parties que nous notons $p(U, A, c)$ (avec $\uparrow d = \{d' \in \mathcal{C} | d' \vdash d\}$). Il est aisé de vérifier que $store(A) \sqcup d \vdash c$ et $vc(d) \subseteq U$ si et seulement si $d \in p(U, A, c)$ et $vc(d) \subseteq U$.

Nous définissons d'abord un système de transitions non annoté dont les règles de réduction sont "libres" – dans le sens où les agents ask, send et receive ne sont plus considérés comme bloquants mais peuvent librement donner lieu à une transition avec la supposition que l'environnement apporte l'information manquante. Un système de transitions annoté est ensuite construit au dessus.

On remarquera que ce système de transitions non annoté ne prend pas en compte l'opérateur de restriction. Nous utilisons en effet un mécanisme différent de celui proposé initialement dans [MPW89] pour assurer la cohérence des noms de canaux. L'idée est ici de travailler avec des agents en forme standard et des variables locales fixées, et de mémoriser un ensemble de variables visibles. Nous définissons pour cela deux restrictions de la congruence structurelle. La première ne considère pas l'opérateur de restriction qui est pris en compte par la seconde. La raison en est que, à l'instar des CC et contrairement au π-calcul, les actions sont ici dépendantes du contexte local (i.e. du store). Le mécanisme des règles (ouverture)/(fermeture) du π-calcul (cf. section 4) n'est donc pas transposable aux DCC. Saraswat et Rinard utilisent un système de transitions dont les états sont des couples formés d'un store et d'un agent. Nous procédons ici en deux temps.

7.2 Bisimulation sur les sites

Nous définissons dans cette section une équivalence de capacité de communication qui ne prend pas en compte l'opérateur *tell*. L'équivalence ainsi obtenue n'a donc d'intérêt que pour les agents qui sont des sites car ceux-ci ne communiquent avec l'environnement que par message.

La congruence structurelle $\equiv^{-\exists}$ sur les agents est la plus petite relation de congruence vérifiant les axiomes suivants :

- $\langle Agents, \|, vrai\rangle$ et $\langle Agents, +, vrai\rangle$ sont des monoïdes commutatifs.
- $c \parallel d \equiv^{-\exists} d \sqcup c$

7.2.1 Système de transitions non annoté

Règles structurelles :

$$\frac{A \longrightarrow A'}{A \parallel B \longrightarrow A' \parallel B} \quad \text{(parallel)}$$

$$\frac{A \rightarrow A'}{A + B \rightarrow A'} \quad \text{(choix)}$$

$$\frac{B \equiv^{-\exists} A \qquad A \to A' \qquad A' \equiv^{-\exists} B'}{B \to B'} \qquad (\text{équiv}^{-\exists})$$

Règles de réduction :

$$\frac{}{ask(c).A \to A} \qquad (\text{ask libre})$$

$$\frac{}{send(u, \lambda\vec{x}c) \longrightarrow true} \qquad (\text{send})$$

$$\frac{}{receive(u, \vec{y}).A \longrightarrow A} \qquad (\text{receive})$$

$$\frac{}{send_{ca}(u, \vec{v}) \longrightarrow true} \qquad (\text{send}_{ca})$$

$$\frac{long(\vec{w}) = long(\vec{v})}{receive_{ca}(u, \vec{v}).A \longrightarrow (\vec{w} = \vec{v} \parallel A)} \qquad (\text{receive}_{ca})$$

$$\frac{d \vdash u = v \qquad d \neq faux \qquad long(\vec{x}) = long(\vec{y})}{d \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).A \to d \parallel c[\vec{y}/\vec{x}] \parallel A} \qquad (\text{communication})$$

$$\frac{p(\vec{x}) :: A \in D \qquad \vec{x} \cap \vec{y} = \emptyset}{p(\vec{y}) \to \exists_{\vec{x}}(\vec{x} = \vec{y} \parallel A)} \qquad (\text{récursion})$$

Dans le cas du π-calcul, les règles correspondant aux règles (communication) et (récursion) sont des règles structurelles. L'observation d'une action de sortie et d'une action d'entrée sur le même nom infère une communication (silencieuse car interne à l'agent observé). Nous ne pouvons pas utiliser ici la même technique car la communication dépend d'une contrainte en composition parallèle qui établit l'égalité des canaux.

Nous conservons le style de sémantique utilisé aux chapitres 3 et 5. Notons que les actions silencieuses supplémentaires ainsi introduites (par la règle (récursion)) n'ont d'incidence que sur la bisimilarité forte. Cela n'influence pas la bisimilatité faible, celle-ci ne tenant pas compte des actions silencieuses.

7.2.2 Système de transitions annoté

Nous définissons un système de transitions annoté qui capture la capacité de communication des sites. Les états sont les couples $\langle G, L, A\rangle$ où G est un ensemble de variables de contraintes et L est un ensemble de variables de canaux (G (resp. L) signifie *global* (resp. *local*)) et A est un agent DCC (sans agents *lien*). Ces ensembles de variables serviront à construire de manière dynamique les ensembles des variables globales (ou visibles) et locales. Les variables visibles sont les variables de contraintes par l'intermédiaire desquelles l'agent est suceptible de recevoir de l'information de l'environnement. Les variables locales sont les variables de canaux qui ont été "fixées" par la règle (exists_{nom}) (cf. ci-desssous). Cet ensemble servira à mémoriser les variables de canaux qui sont locales, la règle (exists_{nom}) supprimant l'opérateur de restriction.

Les annotation sont des agents ou l'action silencieuse τ. Plus précisément, ces agents sont soit des contraintes soit des agents de communication. Soit A un agent, $fv(A)$ (resp. $bv(A)$) dénote l'ensemble des variables libres (resp. liées) de A. En ce qui concerne l'action τ, nous avons $fv(\tau) = bv(\tau) = \emptyset$.

La notion de transition initiale utilisée ci-dessous est celle de la définition 6.2 étendue de manière évidente aux transitions "libres".

Soit $A \longrightarrow B$ une transition du système de transitions annoté et $A' \longrightarrow B'$ sa transition initiale.
Soient G, L deux ensembles de variables.

- Si $A' \longrightarrow B'$ est une règle (communication) ou (récursion) alors $\langle G, L, A\rangle \xrightarrow{\tau} \langle G, L, B\rangle$.

- Si $A' \longrightarrow B'$ est $ask(c).B' \longrightarrow B'$ et $store(A) \vdash c$ alors $\langle G, L, A\rangle \xrightarrow{\tau} \langle G, L, B\rangle$.

- Si $A' \longrightarrow B'$ est $ask(c).B' \longrightarrow B'$ et $store(A) \not\vdash c$ alors $\langle G, L, A\rangle \xrightarrow{p(G,A,c)} \langle G, L, B\rangle$

- Si $A' \longrightarrow B'$ est $send(u, \lambda\vec{x}c).B' \longrightarrow B'$ et $store(A) \vdash u = v$ pour $v \notin L$ alors
  $\langle G, L, A\rangle \xrightarrow{send(v,\lambda\vec{x}c)} \langle G, L, B\rangle$

- Si $A' \longrightarrow B'$ est $receive(u, \vec{y}).B' \longrightarrow B'$, $store(A) \vdash u = v$ pour $v \notin L$ et $store(A) \vdash \vec{z} = \vec{y}$ alors
  $\langle G, L, A\rangle \stackrel{receive(u,\vec{z})}{\longrightarrow} \langle G \cup \vec{y}, L, B\rangle$
- Si $A' \longrightarrow B'$ est $send_{ca}(u, \vec{v}).B' \longrightarrow B'$, $store(A) \vdash u = u'$ pour $u' \notin L$ et :
  S'il exists v' tel que $store(A) \vdash \vec{v} = \vec{v}'$ et $v' \notin L$ alors
  $\langle G, L, A\rangle \stackrel{send_{ca}(u',\vec{v}')}{\longrightarrow} \langle G, L, B\rangle$
  Sinon
  $\langle G, L, A\rangle \stackrel{send_{ca}(u',\vec{v})}{\longrightarrow} \langle G, L/\vec{v}, B\rangle$
- Si $A' \longrightarrow B'$ est $receive_{ca}(u, \vec{v}).B' \longrightarrow (\vec{w} = \vec{v} \parallel B')$, $store(A) \vdash u = u'$ pour $v \notin L$, et $\vec{w} \cap L = \emptyset$ alors
  $\langle G, L, A\rangle \stackrel{receive_{ca}(u',\vec{w})}{\longrightarrow} \langle G, L, B\rangle$

Les règles structurelles du système de transitions annoté sont les suivantes :

$$\frac{x \in V_{co}}{\langle G, L, \exists_x A\rangle \stackrel{\tau}{\longrightarrow} \langle G, L, A\rangle} \qquad (\text{exists}_{var})$$

$$\frac{u \in V_{ca}}{\langle G, L, \exists_u A\rangle \stackrel{\tau}{\longrightarrow} \langle G, L \cup \{u\}, A\rangle} \qquad (\text{exists}_{nom})$$

La congruence structurelle et la règle structurelle définies ce-dessous prennent en compte l'opérateur de restriction.

La congruence structurelle $\equiv^{\exists}$ sur les agents est la plus petite relation de congruence vérifiant les axiomes suivants :

- $\exists_x A \equiv^{\exists} \exists_y A[y/x]$ si y n'est pas libre dans A
  $\exists_x A \parallel B \equiv^{\exists} \exists_x (A \parallel B)$ si x n'est pas libre dans B
  $\exists_x A + B \equiv^{\exists} \exists_x (A + B)$ si x n'est pas libre dans B
  $\exists_x \exists_y A \equiv^{\exists} \exists_y \exists_x A$
  $\exists_x A \equiv^{\exists} A$ si x n'est pas libre dans A

$$\frac{B \equiv^{\exists} A \qquad \langle G, L, A\rangle \stackrel{\alpha}{\longrightarrow} \langle G', L', A'\rangle \qquad A' \equiv^{\exists} B'}{\langle G, L, B\rangle \stackrel{\alpha}{\longrightarrow} \langle G', L', B'\rangle} \qquad (\text{equiv}^{\exists})$$

Les règles (exists$_{var}$) et (exists$_{nom}$) ont pour effet de “fixer” les variables de canaux locales. Nous observons ici le même phénomène que pour la règle (récursion), l'élimination des opérateurs de restriction introduit une action silencieuse. Cette transition supplémentaire, que l'on pourrait qualifier de “parasite”, est cependant unique (l'élimination d'un opérateur de restriction introduit une unique action silencieuse).

Nous définissons maintenant la relation de bisimilarité forte (et close).

Définition 7.1 Une relation quaternaire $\sim \subseteq P_f(V) \times P_f(V) \times Agents \times Agents$ est une *simulation forte* si $\sim (G, L, A, B)$ (nous le noterons $G, L : A \sim B$) implique :

- b1 Si $\langle G, L, A \rangle \xrightarrow{\alpha} \langle G', L', A' \rangle$, α n'est pas un agent $receive_{ca}$ et $(fv(B) \cap L) = \emptyset$ alors il existe un agent B' tel que :
  $\langle G, L, B \rangle \xrightarrow{\alpha} \langle G', L', B' \rangle$ et $G', L' : A' \sim B'$.
- b2 Si $\langle G, L, A \rangle \xrightarrow{receive_{ca}(u,\vec{v})} \langle G', L', A' \rangle$ alors il existe un agent B' tel que :
  $\langle G, L, B \rangle \xrightarrow{receive_{ca}(u,\vec{v})} \langle G', L', B' \rangle$ et $G', L' : A' \sim B'$ ou
  $\langle G, L, B \rangle \xrightarrow{\tau} \langle G', L', B' \rangle$ et $G', L' : A' \sim (B' \parallel send_{ca}(u, \vec{v}))$

□

Remarquons qu'il n'est pas possible de traiter ici les agents *receive*, comme cela est fait pour les agents $receive_{ca}$ ou le préfixe d'entrée du π-calcul, d'une manière plus adaptée à la communication asynchrone (recherche non seulement d'une action identique mais d'une telle action ou d'une similarité en présence d'un consommateur de cette action). La raison en est que nous n'avons pas défini leurs réductions libres selon le mécanisme *early instantiation*. Notons cependant que cela serait possible avec la modification suivante de la règle (receive) :

$$\frac{\vec{y} \subseteq fv(c)}{receive(u, \vec{y}).A \longrightarrow (c \parallel A)}$$

L'introduction de cette règle nécessiterait, en plus des ensembles G et L, d'utiliser dans le système de transitions annoté un ensemble de contraintes locales. Bien que cela ne poserait pas de réel problème, nous ne le faisons pas afin de ne pas alourdir le formalisme.

Définition 7.2 Une relation $\sim$ est une *bisimulation forte* si $\sim$ et sa relation "inverse" $\sim'$ ($G, L : A \sim' B$ *ssi* $G, L : B \sim A$) sont des simulations fortes.
La relation de *bisimilarité forte* est la plus grande bisimulation forte. $\Box$

Définition 7.3 Deux sites A et B sont dits *fortement bisimilaires* si il existe une relation de bisimulation $\sim$ telle que $\emptyset, \emptyset : A \sim B$ $\Box$

7.3 Bisimilarité faible

Nous définissons dans cette section une relation de bisimilarité faible qui permet de ne pas tenir compte des transitions non observables. Nous devons rendre identiques les observations de deux suites de transitions comportant une même action non silencieuse parmi une suite quelconque d'actions silencieuses.

Notons $\Rightarrow$ la clôture réflexive et transitive de $\xrightarrow{\tau}$.
Notons $\stackrel{\alpha}{\Rightarrow}$ la relation $\Rightarrow\xrightarrow{\alpha}\Rightarrow$.

Définition 7.4 La relation de transition faible $\stackrel{\hat{}}{\Rightarrow}$ est la plus petite relation définie par :
Soit A et B deux agents,
Si $\alpha \neq \tau$ et $A \stackrel{\alpha}{\Rightarrow} B$ alors $A \stackrel{\hat{\alpha}}{\Rightarrow} B$.
Si $A \Rightarrow B$ alors $A \stackrel{\hat{\tau}}{\Rightarrow} B$. $\Box$

Les bisimilations et la bisimilarité faibles sont obtenues par remplacement dans la définition 7.1 de $A \xrightarrow{\alpha} B$ par $A \stackrel{\hat{\alpha}}{\Rightarrow} B$.

Définition 7.5 Une relation quaternaire $\sim \subseteq P_f(V) \times P_f(V) \times Agents \times Agents$ est une *simulation faible* si $\sim (G, L, A, B)$ (nous le noterons $G, L : A \sim B$) implique :

b1 Si $\langle G, L, A\rangle \stackrel{\hat{\alpha}}{\Rightarrow} \langle G', L', A'\rangle$, α n'est pas un agent $receive_{ca}$ et $(fv(B) \cap L) = \emptyset$ alors il existe un agent B' tel que :
$\langle G, L, B\rangle \stackrel{\hat{\alpha}}{\Rightarrow} \langle G', L', B'\rangle$ et $G', L' : A' \sim B'$.

b2 Si $\langle G, L, A\rangle \stackrel{\widehat{receive_{ca}(u,\vec{v})}}{\Rightarrow} \langle G', L', A'\rangle$ alors il existe un agent B' tel que :
$\langle G, L, B\rangle \stackrel{\widehat{receive_{ca}(u,\vec{v})}}{\Rightarrow} \langle G', L', B'\rangle$ et $G', L' : A' \sim B'$ ou
$\langle G, L, B\rangle \stackrel{\hat{\tau}}{\Rightarrow} \langle G', L', B'\rangle$ et $G', L' : A' \sim (B' \parallel send_{ca}(u, \vec{v}))$

□

Définition 7.6 Une relation $\sim$ est une *bisimulation faible* si $\sim$ et sa relation "inverse" $\sim'$ ($G, L : A \sim' B$ *ssi* $G, L : B \sim A$) sont des simulations faibles.
La relation de *bisimilarité faible* $\dot{\sim}$ est la plus grande bisimulation faible. □

Définition 7.7 Deux sites A et B sont dits *faiblement bisimilaires* (noté $A \dot{\sim} B$) si $\emptyset, \emptyset : A \dot{\sim} B$ □

7.4 Encodage du π-calcul asynchrone

Nous donnons dans cette section un encodage du π-calcul asynchrone dans les DCC. Nous prouvons que cet encodage est complètement adéquat[1].

Soit N l'ensemble des noms du π-calcul. Afin de simplifier le formalisme manipulé, nous supposerons $N \subseteq V_{ca}$.

L'encodage du π-calcul asynchrone dans les DCC est donné par la relation $[\![\,]\!]$ définie comme suit.

$[\![\mathbf{0}]\!] = vrai$
$[\![\tau.A]\!] = ask(vrai).[\![A]\!]$
$[\![\bar{u}v]\!] = send_{ca}(u, v)$
$[\![u(v).A]\!] = \exists_v receive_{ca}(u, v).[\![A]\!]$
$[\![A|B]\!] = [\![A]\!] \parallel [\![B]\!]$
$[\![A + B]\!] = [\![A]\!] + [\![B]\!]$
$[\![[u = v]A]\!] = ask(u = v).[\![A]\!]$
$[\![p(u_1, ..., u_n)]\!] = p(u_1, ..., u_n)$
$[\![(\nu x)A]\!] = \exists_x [\![A]\!]$

Cela ne générant pas de confusion, nous noterons $\dot{\sim}$ les relations de bisimilarité faibles du π-calcul et des DCC.

Remarquons que l'encodage d'un agent du π-calcul est trivialement un site (définition 6.4) puisqu'il ne comporte aucune variable de contrainte.

[1] fully abstract

Proposition 7.1 Soient A et B deux agents du π-calcul asynchrone.
$A \dot{\sim} B$ si et seulement si $[\![A]\!] \dot{\sim} [\![B]\!]$. $\square$

La preuve de cette proposition se trouve en Annexe A. Cette preuve manipule des formalismes plutôt lourds. La difficulté majeure est due à la différence entre le mécanisme de communication du π-calcul qui fonctionne par renommage du nom de réception et celui des DCC qui introduit une contrainte d'égalité entre les variables et un opérateur de restriction sur la variables de réception. L'idée de la preuve est d'établir une bijection entre les dérivations faibles pour un agent du π-calcul et son encodage DCC via un encodage des actions.

Notons que nous encodons ainsi les variables du π-calcul avec des variables logiques (celles des DCC). Cela n'avait, à notre connaissance, pas encore été effectué.

7.5 Bisimulation sur les agents DCC : perspectives et travaux futurs

Définir une sémantique de bisimulation sur l'ensemble des agents DCC et non plus sur les sites seulement ne semble pas poser de réel problème. Inclure l'observation des agents *tell* dans les systèmes de transitions annotés et non annotés précédents peut s'effectuer en s'inspirant de [SR90] dans lequel l'équivalence d'observation proposée comprend ces agents.

Le cas des agents *lien* semble plus délicat mais doit pouvoir être géré par l'intermédiaire de l'opérateur de point fixe dont nous parlons à la section 5.5 et qui permettrait de calculer le store d'un agent comportant des sous agents *lien*. Cela reste cependant à développer. Un tel travail nécessiterait aussi de définir la notion de simulation d'une manière adaptée à la communication asynchrone (cf. notre discussion à ce sujet au chapitre 4) pour les agents *receive* et les agents *ask* (rappelons que nous ne l'avons fait ici que pour les agents $receive_{ca}$).

De Boer et Palamidessi ([BP91]) définissent une relation de congruence sur les CC qui est plus forte que celle de ([SR90]). Ils utilisent des structures différentes de celles employés dans [SR90] et dans ce présent chapitre. [BP91] effectue des suppositions concernant les actions que peut produire l'environnement tandis que les structures que nous utilisons représentent les suites d'actions dont le processus a besoin pour s'exécuter (i.e. la supposition minimale dont le processus a besoin). Un point intéressant est que De Boer et Palamidessi proposent un traitement particulier de l'opérateur de choix (nous renvoyons à ce sujet le lecteur intéressé à [BP91]) qui adapte la congruence obtenue au cas de la communication asynchrone (ce n'est pas le cas dans [SR90]). Une perspective très intéressante de travaux futurs est la comparaison de la congruence de [BP91] avec la bisimilarité sur les DCC restreinte au sous langage CC, ce qui devrait apporter une meilleure compréhension des équivalences d'observation pour les langages à communication asynchrone.

Chapitre 8

Conclusion

Nous présentons dans cette thèse un paradigme de calcul concurrent par contraintes issu des CC (langages Concurrents avec Contraintes). La principale motivation de notre travail est la modélisation de systèmes distribués. Les CC, apparus à la fin des années 80, forment un paradigme de calcul concurrent par contraintes attractif mais qui se prête mal à la distribution. Construire un langage distribué directement basé sur les CC nécessiterait d'effectuer de la résolution distribuée de contraintes, ce qui reste à l'heure actuelle un problème difficile et mal maîtrisé. La plupart des modèles de calcul distribué proposent un mécanisme de communication par messages, associé à un mécanisme (impératif) de calcul local. Il semble naturel de proposer un modèle de calcul distribué dans lequel chaque site est un CC, et d'introduire de la communication par messages pour réaliser l'échange d'information entre sites.

Le π-calcul est un modèle de calcul qui modélise la communication par messages (et seulement celle-ci). Il est issu du CCS (Calculus of Communicating Systems) dans lequel il introduit la possibilité de reconfiguration dynamique du réseau de communication. Nous nous en inspirons pour définir un paradigme concurrent par contraintes (appelé MCC : Message-based CC) qui intègre une notion de communication par messages. Les contraintes permettent une modélisation logique de la synchronisation due à ce mécanisme de communication et il s'avère que la communication $ask/tell$ des CC et la communication par messages s'intègrent d'une manière très homogène, les canaux de communication étant modélisés par des variables du système de contraintes.

Les MCC forment un paradigme très général qui permet une utilisation quelconque des deux modes de communication. Nous définissons sur les MCC une notion de *distribuabilité*. Intuitivement, un programme MCC est distribuable s'il est la composition parallèle d'agents qui ne communiquent que par messages. L'idée de base permettant de construire des programmes MCC distribuables est de limiter la communication *ask/tell* (qui nécessite d'effectuer de la résolution de contraintes) au niveau local. Nous définissons à cet effet un sous langage des MCC appelé DCC (Distributed CC). Les DCC utilisent une notion de typage des variables qui permet de séparer syntaxiquement les variables utilisées dans des contraintes (variables de contraintes) et les variables utilisées comme canaux de communication (variables de canaux). Nous définissons une notion de *site* sur les agents DCC et un résultat important de cette thèse est qu'une composition parallèle de sites DCC est distribuable.

Nous définissons aussi un mécanisme de communication du résultat d'un calcul local (agents *lien*) qui n'a, à notre connaissance, pas d'équivalent dans les travaux liés à la distribution ou aux CC. L'idée est de créer un lien "statique" entre certaines variables de deux sites, l'un étant le site émetteur (site sur lequel s'exécute le calcul local) et l'autre le site récepteur. L'information produite localement sur ces variables du site émetteur est transmise aux variables du site récepteur sur lesquelles porte le lien ; cette propagation d'information est donc unidirectionnelle.

La sémantique des MCC et des DCC est décrite par une sémantique opérationelle de type SOS. Celle-ci permet une analyse de haut niveau grace à une relation de bisimilarité. La notion de bisimulation joue un rôle important en théorie de la concurrence. C'est sur cette notion que reposent les sémantiques du CCS et du π-calcul. Nous définissons une relation de bisimilarité sur une restriction (nous n'observons pas les agents *tell* et nous ne prenons pas en compte les agents *lien*) des agents DCC. Nous encodons le π-calcul dans les DCC. Cet encodage est prouvé être complètement adéquat. Des travaux restent ici à faire, en particulier la rédaction précise et complète de la relation de bisimilarité sur les agents DCC, ainsi que la comparaison avec d'autres travaux concernant les équivalences d'observation dans les CC (en particulier [BP91]).

Un autre travail intéressant serait l'introduction dans les MCC et les DCC d'un opérateur de choix gardé dont l'intérêt apparaît évident dans le cadre d'une implantation indéterministe.

Comme dans le cas du π-calcul, la communication par messages est indéterministe et l'utilisation des canaux est très générale. Implanter les DCC de manière distribuée nécessiterait d'affecter chaque canal de communication à un site particulier qui jouerait alors le rôle d'un buffer. Une autre possibilité serait de limiter la liberté d'utilisation des canaux de communication à une topologie de réseau particulière. Le π-calcul n'est d'ailleurs pas à proprement parler orienté vers la distribution et des travaux sont menées à ce sujet afin d'y intégrer une notion de *localité* ([FG96, FGL+96]). Ces travaux nous offrent des pistes intéressantes de développements futurs.

Les progrès au niveau du matériel informatique – *hardware* – ont été fulgurants. Les systèmes informatiques qui étaient il y a vingt ans centralisés et financièrement coûteux sont aujourd'hui parallèles et distribués, reliés en réseaux et d'un prix de revient beaucoup plus faible. Les progrès dans le domaine des langages de programmation n'ont pas suivi la rapidité de cette évolution et la plupart des langages utilisés à l'heure actuelle ont été conçus pour des systèmes séquentiels et centralisés. La recherche concernant le parallélisme, la concurrence et la distribution apparaît donc d'un intérêt évident et les technologies de l'Internet, pour ne citer que cet exemple ô combien médiatique, ont besoin d'idées nouvelles dans le domaine de la programmation. D'autre part, les langages dits *déclaratifs* offrent une perspective très intéressante de programmation plus simple et mieux maîtrisée que celle issue des langages impératifs. Peu de travaux ont été entrepris en ce qui concerne la distribution de ces langages. Dans le cadre de la programmation en logique, citons [PN84, BC93]. Il sagit plutôt d'approches de style *blackboard*, la communication s'effectuant non pas par messages mais par unification de deux termes. On y trouve cependant l'idée d'utiliser un ensemble de noms de canaux de communication. On peut voir les MCC comme une continuation de ces travaux (avec cependant un mode de communication assez différent), les contraintes nous permettant une modélisation logique de la communication par messages. Nous espérons par cette thèse apporter notre contribution à ce problème en proposant un paradigme de calcul qui intègre de manière homogène communication par messages et calcul concurrent par contraintes.

Le modèle CC est basé sur une mémoire partagée (*store*) et l'apport de cette thèse à été de dévelloper un cadre plus général adapté aux systèmes à mémoire distribuée (réseaux d'ordinateurs), plus adapté à l'informatique actuelle (Internet,...). Les contraintes permettent une modélisation logique de la communication par messages. Les fondements théoriques de ce paradigme de calcul sont ainsi très homogènes. Nous définissons un sous langage permettant la construction de programmes décomposés en *sites* qui ne communiquent que par messages. De tels programmes ouvrent la possibilité d'une exécution distribuée évitant le recours à la résolution distribuée de contraintes. L'étude précise d'une implantation distribuée des DCC reste cependant à faire.

Bibliographie

[ACS96] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *proc. of Concur'96*, pages 147–162, 1996.

[And91] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings, 1991.

[AS88] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. In N. Gehani and A.D. McGettrick, editors, *Concurrent Programming*, pages 3–69. Addison-Wesley, 1988.

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

[BC91] A. Brogi and P. Ciancarini. The concurrent language shared prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[BC93] A. Brogi and P. Ciancarini. Distributed logic programming. *Journal of Logic Programming*, 15(4):295–335, 1993.

[BFB$^+$95] M. Bergère, G. Ferrand, F. le Berre, B. Malfon, and A. Tessier. La programmation logique avec contraintes revisitée en terme d'arbres de preuve et de squelettes. Technical Report RR 95-06, Université d'Orléans, LIFO, Février 1995.

[BFR93] M. Bergère, G. Ferrand, and J.-H. Réty. Independence of the computation rule for SLD-resolution with delay. Technical Report RR 93-9, Université d'Orléans, LIFO, Novembre 1993.

[BFR95] M. Bergère, G. Ferrand, and J.-H. Réty. Indépendance de la règle de calcul pour la résolution SLD avec retard. *Technique et Science Informatiques*, 14(6):693–712, 1995.

[BGJK96a] L. Brim, D. Gilbert, J.M. Jacquet, and M. Kr̀etínský. New versions of ask and tell for synchronous communication in concurrent constraint programming. Technical Report 1996/03, City University, Dept. of Computer Science, London, 1996.

[BGJK96b] L. Brim, D. Gilbert, J.M. Jacquet, and M. Kr̀etínský. A process algebra for synchronous concurrent constraint programming. In *proceedings of ALP'96*, volume 1106 of *LNCS*, pages 165–178. Springer-Verlag, 1996.

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. Confluence and Concurrent Constraint Programming. Proceedings of the Post-Conference Workshop, ICLP , 1994.

[BHMR97] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. Partial order and contextual net semantics for atomic and locally atomic CC programs. *Science of Computer Programming, Special issue on Concurrent Constraint Programming*, 1997. To appear.

[BKPR91] F.S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Semantic models for concurrent logic languages. *Theoretical Computer Science*, 86:3–33, 1991.

[BKPR93] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Ruten. Non-momotonic concurrent constraint programming. In *Proc. of ILPS'93*. MIT Press, 1993.

[Bou92] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Mai 1992.

[BP91] F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proc. of TAPSOFT/CAAP, LNCS 493*, pages 296–319. Springer-Verlag, 1991.

[BP94] F.S. de Boer and C. Palamidessi. From concurrent logic programming to concurrent constraint programming. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 55–113. Oxford University Press, 1994.

[BPP95] F.S. de Boer, A. di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151:37–78, 1995.

[BPP96] F.S. de Boer, A. di Pierro, and C. Palamidessi. An algebraic perspective of constraint logic programming. *Journal of Logic and Computation*, 1996. To appear.

[BS96] M. Boreale and D. Sangiorgi. Some congruence properties for π-calculus bisimilarities. Technical Report 2870, INRIA, 1996.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), 1989.

[CC93] C. Codognet and P. Codognet. A general semantics for concurrent constraint languages and their abstract interpretation. In Manfred Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93, St. Petersburg*, pages 43–50, 1993.

[CCRS89] P. Codognet, J. Chassin de Kergommeaux, P. Robert, and J.-C. Syre. Une revue des modèles de programmation parallèle. *Techniques et Sciences Informatiques*, (3 & 4), 1989.

[CFMW93] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. proc. of the 20th International Colloquium on Automata, Languages, and Programming, 1993.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM 32(4)*, April 1989.

[Cha97] P. Chambre. *Contribution à la validation de programmes concurrents avec contraintes.* PhD thesis, Université d'Orléans, Janvier 1997.

[Cia94] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251–284, 1994.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[DGP92] J. Darlington, Y. Guo, and H. Pull. A new perspective on integrating functional and logic programming. In *proc. of the International Conference on Fifth Generation Computer Systems*, pages 682–693. ICOT, 1992.

[DM93] P. Deransard and J. Maluszinski. *A Grammatical View of Logic Programming.* MIT press, 1993.

[Doe94] K. Doets. *From Logic to Logic Programming.* MIT Press, 1994.

[Ede85] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation (1)*, pages 31–46, 1985.

[EN86] U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical Report DAIMI PB-208, University of Aarhus, 1986.

[Fag96] F. Fages. *Programmation Logique par Contraintes.* Ellipses, Paris, 1996.

[Fer91] G. Ferrand. Basics concepts of logic programming and prolog (tutorial ICLP'91). Technical report, INRIA, 1991.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.

[FGMP95] M. Falaschi, M. Gabrielli, K. Marriott, and C. Palamidessi. Confluence and concurrent constraint programming. In *proc. of the fourth International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 936 of *LNCS*, pages 531–545. Springer-Verlag, 1995.

[Her88] R. Herken, editor. *The Universal Turing machine, A Half-Century survey.* Oxford University Press, 1988.

[HMT71] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras (Part. 1).* North-Holland, 1971.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *proc. of ECOOP'91*, pages 133–147. Springer-Verlag, LNCS 512, 1991.

[HT92] K. Honda and M. Tokoro. On asynchronous communication sémantics. In *Object-based concurrent computing, SLNCS 612*, 1992.

[Hue80] G. Huet. Confluent reductions, abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27:797–821, 1980.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *proc. of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[JM92] J.-M. Jacquet and L. Monteiro. Communicating clauses: towards synchronous communication in contextual logic programming. In *proc. of JICSLP*, 1992.

[JSS91] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. In *proc. of LICS*, June 1991.

[Kow79] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[KS90] K. Kahn and V. Saraswat. Actors as a special case of concurrent constraint programming. In *OOPSLA/ECOOP'90*, 1990.

[KW93] P. Kacsuk and M.J. Wise, editors. *Implementations of Distributed Prolog.* Series in parallel computing. Wiley, 1993.

[Leu93] H.-F. Leung. *Distributed Constraint Logic Programming.* Series in Computer Science. World Scientific, 1993.

[LL90] L. Lamport and N. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[LM92] C. Laneve and U. Montanari. Mobility in the cc-paradigm. In *proc. of Mathematics Foundations of Computer Science*, 1992.

[LMM88] J.L. Lassez, M.J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgann Kaufmann, Los Altos, 1988.

[Mah87] M. Maher. Logic semantics for a class of commited-choice programs. In *proc. of 4^{th} International Conference on Logic Programming*. MIT press, May 1987.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] R. Milner. Operational and algebraic semantics of concurrent processes. *Handbook of Theoretical Computer Science*, pages 1201–1243, 1990.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edimburgh, 1991.

[Mil92] R. Milner. Functions as processes. *Mathematical Structures in Computer Science, vol 2*, pages 119–141, 1992.

[MPW89] R. Milner, J.G. Parrow, and D.J. Walker. A calculus of mobile processes, parts I and II. Technical Report ECS-LFCS-89-85 and -86, LFCS, Edinburgh University, 1989.

[MR91] U. Montanari and F. Rossi. True Concurrency in Concurrent Constraint Programming. Technical report, Dipartimento di Informatica, Università di Pisa, 1991.

[Nie94] J. Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen.* Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, 66041 Saarbrücken, Germany, December 1994. In German.

[Nie95] J. Niehren. Functional computation as concurrent computation. Research Report RR-95-14, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D66123 Saarbrücken, Germany, November 1995. shorter version appeared in: *Proceedings of POPL'96*, The ACM Press.

[NS94] Joachim Niehren and Gert Smolka. A confluent relational calculus for higher-order programming with constraints. In Jean-Pierre

Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *LNCS*, pages 89–104, München, Germany, September 1994.

[Pie96] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. Available electronically, 1996.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[Plo83] G. Plotkin. Domains. TEX-ed edition prepared by Y. Kashiwagi and H. Kondoh, 1983. Courses notes.

[PN84] L. Pereira and R. Nasr. Delta-prolog: A distributed logic programming language. In *proc. of International Conference on Fifth Generation Computer systems*, pages 283–291. ICOT, 1984.

[PP96] C. Palamidessi and E. Pimentel. Comparing the expressive power of π-calculus and concurrent constraint programming. Unpublished, 1996.

[PSSS91] P. Panangaden, V. A. Saraswat, P. J. Scott, and R. A. G. Seely. What is a constraint system. Technical report, Xerox PARC, December 1991.

[Rét95a] J.-H. Réty. The notion of floundering for SLDNF-resolution revisited. In *proc. of Programming Languages: Implementation, Logics and Programs, LNCS 982*, pages 380–396, 1995.

[Rét95b] J.-H. Réty. Redéfinir l'enlisement pour la résolution SLDNF. In *proc. of Journées françaises de Programmation en Logique*, 1995.

[Sar88] V.A. Saraswat. A somewhat logical formulation of CLP synchronisation primitives. In *proc. LP 88*. MIT press, August 1988.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming*. Phd thesis, Carnegie-Mellon University, 1989.

[Sar92a] V.A. Saraswat. The category of constraint systems is cartesian closed. In *proc. 7th IEEE Symp. on Logic in Computer Science*, 1992.

[Sar92b] V.A. Saraswat. Concurrent constraint programming: A brief survey. Technical report, Xerox PARC, August 1992.

[Sar93] V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Sco82] D.S. Scott. Domains for denotational semantics. In *proc. of ICALP*, 1982.

[Sha89] E. Shapiro. The Family of Concurrent Logic Programming Languages. Technical Report CS89-08, Weizmann Institute of Science, Rehovot, Israel, May 1989.

[SKL89] Vijay Saraswat, Kenneth Kahn, and Jacob Levy. Programming in Janus. Technical report, Xerox PARC, 1989.

[SKL90] V.A. Saraswat, K. Kahn, and J. Levy. Janus: a step towards distributed constraint programming. In *proc. of the North American Conference on Logic Programming*, 1990.

[Smo94] Gert Smolka. A Foundation for Concurrent Constraint Programming. In *proc. of 1st International Conference on Constraints in Computational Logics*, volume 845 of *LNCS*, Munich, Germany, September 1994. Invited Talk.

[SR90] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *proc. of Seventeenth ACM Symposium on Principles of Programming Languages*, 1990.

[SRP91] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *proc. eighteen annual ACM symposium on Principles of Programming Languages*, pages 333–353, 1991.

[VP96] B. Victor and J. Parrow. Constraints as processes. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1996.

Annexe A

Preuve de la proposition 7.1

Considérons l'encodage suivant des actions du π-calcul asynchrone dans les actions des DCC :
$[\![\tau]\!] = \tau$
$[\![\bar{u}v]\!] = [\![\bar{u}(v)]\!] = send_{ca}(u, v)$
$[\![\bar{u}v]\!] = receive_{ca}(u, v)$

Nous supposerons que V_{ca}/N est non vide et comporte une infinité d'éléments. La raison d'être de cette hypothèse est que le π-calcul gère la réception d'un nom par renommage (règle (entrée-act)) tandis que dans les DCC, il y a ajout d'une contrainte d'égalité entre la variable de canal transmise et celle de réception, et d'un opérateur de restriction sur cette dernière. Cela simplifiera les preuves que de considérer qu'après réduction, la variable de réception n'est pas un nom du π-calcul. Cela n'entraine aucune perte de généralité car la règle (equiv$^{\exists}$) permet le renommage des noms liés.

Nous pouvons aussi supposer que les agents du π-calcul que nous manipulons sont en forme prénexe. Cela n'entraine pas de perte de généralité car pour tout agent (du π-calcul), il existe un (au moins) agent structurellement équivalent en forme prénexe. De plus, il est immédiat de constater que, par le règle (équiv), un agent et sa forme prénexe possèdent les mèmes dérivations pour la relation de transition annotée $\dot{\rightarrow}$, et donc aussi pour la relation de transition annotée faible $\hat{\dot{\Rightarrow}}$.

Considérons un ensemble D de déclarations de procédures du π-calcul.

Lemme A.1 Soit $(\nu u_1, ..., u_n)A$ un agent du π-calcul.
Nous supposons que A ne contient aucun opérateur de restriction.
Si $(\nu u_1, ..., u_n)A \overset{\hat{\alpha_0}}{\Rightarrow} A_1 \overset{\hat{\alpha_1}}{\Rightarrow} A_2$... alors
$\langle \emptyset, \emptyset, [\![(\nu u_1, ..., u_n)A]\!]\rangle \overset{[\hat{\alpha_0}]}{\Rightarrow} \langle \emptyset, L_1, \tilde{A}_1\rangle \overset{[\hat{\alpha_1}]}{\Rightarrow} \langle \emptyset, L_2, \tilde{A}_2\rangle$... avec,
pour tout i, si $\alpha_i = \bar{u}(v)$ alors $v \in L_i$ et,
pour tout i, si $\alpha_i = \bar{u}v$ alors $v \notin L_i$. $\square$

preuve :
Par induction sur la longueur de la dérivation et sur la syntaxe du π-calcul.

0. On a par la règle (exists$_{nom}$) :
$\langle \emptyset, \emptyset, [\![(\nu u_1, ..., u_n)A]\!]\rangle \Rightarrow \langle \emptyset, \{u_1, ..., u_n\}, [\![A]\!]\rangle$

Par la suite, nous noterons U l'ensemble $\{u_1, ..., u_n\}$

1. Par réflexivité des relations $\Rightarrow$, on a pour tout agent B
$B \Rightarrow B$ et
$\langle \emptyset, L, [\![B]\!]\rangle \Rightarrow \langle \emptyset, L, [\![B]\!]\rangle$.
Cela inclut, en particulier, le cas de l'agents $\mathbf{0}$.

2. Cas $A = \tau.B$. On a
$(\nu U)\tau.B \Rightarrow (\nu U)B$ et
$\langle \emptyset, U, ask(vrai).[\![B]\!]\rangle \Rightarrow \langle \emptyset, U, [\![B]\!]\rangle$

3. Cas $A = u(v).B$
3.α. $u \in U$ ou $v \in U$. Ce cas est traité par 1. car il n'existe alors aucune transition pour $(\nu U)A$ autre que $(\nu U)A \Rightarrow (\nu U)A$. De même, il n'existe alors aucune transition pour $\langle \emptyset, U, [\![A]\!]\rangle$ autre que $\langle \emptyset, U, [\![A]\!]\rangle \Rightarrow \langle \emptyset, U, [\![A]\!]\rangle$
3.β. $u, v \notin U$. On a :
$(\nu U)u(v).B \overset{\hat{uw}}{\Rightarrow} (\nu U)B[w/v]$, avec $w \notin U$, et
$\langle \emptyset, U, \exists_v receive_{ca}(u, v).[\![B]\!]\rangle \overset{receive\hat{_{ca}}(u,w)}{\Rightarrow}$
$\langle \emptyset, U, \exists_\mu(\mu = w \parallel [\![B]\!][\mu/v])\rangle$ avec $\mu \in V_{ca}/N$.
(On renomme v par une variable "fraiche" μ qui n'appartient pas à N).

4. Cas $A = \bar{u}v$.
4.α. $u \in U$. Ce cas est traité par 1.
4.β. $u, v \notin U$. On a :
$(\nu U)\bar{u}v \overset{\hat{\bar{u}v}}{\Rightarrow} (\nu U)\mathbf{0}$ et,
$\langle \emptyset, U, send_{ca}(u, v)\rangle \overset{send\hat{_{ca}}(u,v)}{\Rightarrow} \langle \emptyset, U, vrai\rangle$

car $v \notin U$.
De plus, $vrai = [\![\mathbf{0}]\!]$.
4.γ $v \in U$ et $u \notin U$. On a :
$(\nu U)\bar{u}v \overset{\hat{\bar{u}(v)}}{\Rightarrow} (\nu U/v)\mathbf{0}$ et,
$\langle \emptyset, U, send_{ca}(u,v)\rangle \overset{send_{ca}\hat{}(u,v)}{\Rightarrow} \langle \emptyset, U/v, vrai\rangle$.
On a bien $v \in U$.

Nous avons : si $(\nu U)A \overset{\hat{\alpha_0}}{\Rightarrow} (\nu U')A'$ et
$\langle \emptyset, \emptyset, [\![(\nu U)A]\!]\rangle \overset{[\hat{\alpha_0}]}{\Rightarrow} \langle \emptyset, L', \tilde{A}'\rangle$
alors soit $\tilde{A}' = [\![A']\!]$ et $L' = U'$,
soit $\tilde{A}' = \exists_{\vec{\mu}}(\vec{\mu} = \vec{u} \parallel [\![A']\!][\vec{\mu}/\vec{v}])$, avec $\vec{\mu} \cap N = \emptyset$ (on renomme avec des variables fraiches), et $L' = U'$
Dans ce dernier cas, on a la réduction :
$\tilde{A}' \Rightarrow \langle \emptyset, L' \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel [\![A']\!][\vec{\mu}/\vec{v}]\rangle$.
Nous devons donc effectuer l'induction avec des états DCC qui sont :
soit $\langle \emptyset, U, [\![A]\!]\rangle$,
soit $\langle \emptyset, U \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel [\![A]\!][\vec{\mu}/\vec{v}]\rangle$.
U est l'ensemble des variables qui sont des noms et qui sont liés dans l'agent du π-calcul, $\vec{\mu}$ est l'ensemble des variables qui ne sont pas des noms ($\vec{\mu} \cap N = \emptyset$) et qui sont apparus au cours du calcul.

Cela nous permet de limiter la suite de cette preuve au cas général d'une dérivation de longueur n, qui inclut le cas $n = 1$. Nous devrons considérer à chaque étape le cas où l'agent DCC est de type $[\![A']\!]$ et celui où il est de type $(\vec{\mu} = \vec{u} \parallel [\![A']\!][\vec{\mu}/\vec{v}])$.

Nous devons tout d'abord reconsidérer les cas 2 à 4 ci-dessus (le cas 1 est trivial) avec $[\![A']\!]$ de type $(\vec{\mu} = \vec{u} \parallel [\![A']\!][\vec{\mu}/\vec{v}])$. Nous omettons les cas simples qui se réduisent au cas 1.

2bis. Cas $A = \tau.B$. On a
$(\nu U)\tau.B \Rightarrow (\nu U)B$ et
a. $\langle \emptyset, U \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel ask(vrai).[\![B]\!][\vec{\mu}/\vec{v}]\rangle$
$\Rightarrow \langle \emptyset, U \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel [\![B]\!][\vec{\mu}/\vec{v}]\rangle$

3bis. Cas $A = u(v).B$ et $u, v \notin U$ On a :
$(\nu U)u(v).B \overset{\hat{uw}}{\Rightarrow} (\nu U)B[w/v]$, avec $w \notin U$, et
$\langle \emptyset, U \cup \vec{\mu}', \vec{\mu}' = \vec{u}' \parallel \exists_v receive_{ca}(u,v).[\![B]\!][\vec{\mu}'/\vec{v}']\rangle$

$\stackrel{\hat{receive_{ca}(u,w)}}{\Rightarrow} \langle \emptyset, U \cup \vec{\mu}', \vec{\mu}' = \vec{u}' \parallel \exists_\mu(\mu = w \parallel [\![B]\!][\mu/v][\vec{\mu}'/\vec{v}']) \rangle$ avec $\mu \in V_{ca}/N$ et $\mu \notin \vec{\mu}'$.

$\Rightarrow \langle \emptyset, U \cup \vec{\mu}' \cup \vec{\mu}, \vec{\mu}' = \vec{u}' \parallel \mu = w \parallel [\![B]\!][\mu/v][\vec{\mu}'/\vec{v}'] \rangle$

L'agent $\vec{\mu}' = \vec{u}' \parallel \mu = w \parallel [\![B]\!][\mu/v][\vec{\mu}'/\vec{v}']$ est structurellement équivalent à un agent de la forme $\vec{\mu}'' = \vec{u}'' \parallel [\![B]\!][\vec{\mu}''/\vec{v}'']$. $\vec{\mu}'' = \vec{u}''$ étant la conjonction de $\vec{\mu}' = \vec{u}'$ et $\mu = w$, et $[\vec{\mu}''/\vec{v}'']$ étant la composition de renommages $[\mu/v][\vec{\mu}'/\vec{v}']$.

4bis. Cas $A = \bar{u}v$.

4bis.β. $u, v \notin U$. On a :

$(\nu U)\bar{u}v \stackrel{\bar{u}v}{\Rightarrow} (\nu U)\mathbf{0}$ et,

$\langle \emptyset, U \cup \vec{\mu}', \vec{\mu}' = \vec{u}' \parallel send_{ca}(u,v)[\vec{\mu}'/\vec{v}'] \rangle \stackrel{\hat{send_{ca}(u,v)}}{\Rightarrow} \langle \emptyset, U \cup \vec{\mu}', \vec{\mu}' = \vec{u}' \parallel vrai \rangle$

car $v \notin U$.

4bis.γ. $v \in U$ et $u \notin U$. On a :

$(\nu U)\bar{u}v \stackrel{\hat{\bar{u}(v)}}{\Rightarrow} (\nu U/v)\mathbf{0}$ et,

$\langle \emptyset, U \cup \vec{\mu}', \vec{\mu}' = \vec{u}' \parallel send_{ca}(u,v) \rangle \stackrel{send_{ca}(\hat{u,v})[\vec{\mu}'/\vec{v}']}{\Rightarrow} \langle \emptyset, U/v \cup \vec{\mu}', \vec{\mu}' = \vec{u}' \parallel vrai \rangle$.

On a bien $v \in U$.

Par la suite, les cas notés a. (resp. b.) correspondent à un état DCC dont l'agent est de type $[\![A']\!]$ (resp. $(\vec{\mu} = \vec{u} \parallel [\![A']\!][\vec{\mu}/\vec{v}])$).

5. Cas $A = [u = v].B$.

5.*alpha* $u \neq v$. Ce cas est traité par 1.

5.*beta* $u = v$

Supposons $(\nu U)A \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U)(\nu U')B'$, avec $B \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U')B'$ et

$\langle \emptyset, U, [\![B]\!] \rangle \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}' \rangle$ (dans le cas où $\tilde{B}' = [\![B']\!]$ on a $\vec{\mu}' = \emptyset$). On a :

a. $\langle \emptyset, U, ask(u = u).[\![B]\!] \rangle$

$\stackrel{\tau}{\rightarrow} \langle \emptyset, U, [\![B]\!] \rangle$

$\stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}' \rangle$

b. $\langle \emptyset, U \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel (ask(u = v).[\![B]\!])[\vec{\mu}/\vec{u}] \rangle$

On conclut comme précédemment si $\vec{\mu} = \vec{u} \vdash u = v[\vec{\mu}/\vec{u}]$ (Sinon il n'y a aucune transition autre que $\Rightarrow$ ni pour le π-calcul ni pour les DCC) :

$\stackrel{\tau}{\rightarrow} \langle \emptyset, U \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel [\![B]\!][\vec{\mu}/\vec{u}] \rangle$

$\stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \vec{\mu} = \vec{u} \parallel \tilde{B}'[\vec{\mu}/\vec{u}] \rangle$

6. Cas $A = p(\vec{u})$ avec $p(\vec{v}) :: B$

supposons $(\nu U)A \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U)(\nu U')B'$, avec $B[\vec{u}/\vec{v}] \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U')B'$ et $\langle \emptyset, U \cup \vec{\mu}, \exists_{\vec{\mu}}(\vec{u} = \vec{\mu} \parallel [\![B]\!][\vec{\mu}/\vec{v}]\rangle \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}'\rangle$. On a :

a. $\langle \emptyset, U, [\![p(\vec{u})]\!]\rangle$

$\quad \stackrel{\tau}{\rightarrow} \langle \emptyset, U, \exists_{\vec{\mu}}(\vec{u} = \vec{\mu} \parallel [\![B]\!][\vec{\mu}/\vec{v}]\rangle$

$\quad \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}'\rangle$.

b. $\langle \emptyset, U \cup \vec{\mu}'', \vec{\mu}'' = \vec{w} \parallel [\![p(\vec{u})]\!][\vec{\mu}''/\vec{w}]\rangle$

$\quad \stackrel{\tau}{\rightarrow} \langle \emptyset, U \cup \vec{\mu}'', \vec{\mu}'' = \vec{w} \parallel \exists_{\vec{\mu}}(\vec{u} = \vec{\mu} \parallel [\![B]\!][\vec{\mu}/\vec{v}])[\vec{\mu}''/\vec{w}]\rangle$

On conclut comme précédemment :

$\quad \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}'' \cup \vec{\mu}', \vec{\mu}'' = \vec{w} \parallel (\vec{u} = \vec{\mu} \parallel \tilde{B}'[\vec{\mu}/\vec{v}][\vec{\mu}''/\vec{w}]\rangle$.

7. Cas $A = B + C$.

Supposons $(\nu U)A \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U)(\nu U')B'$, avec $B \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U')B'$ et $\langle \emptyset, U, [\![B]\!]\rangle \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}'\rangle$. On a :

a. $\langle \emptyset, U, [\![B + C]\!]\rangle \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}'\rangle$.

(Par règle (choix) des DCC).

b. $\langle \emptyset, U \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel [\![B + C]\!][\mu/\vec{u}]\rangle$

$\quad \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \mu \cup \vec{\mu}', \vec{\mu} = \vec{u} \parallel \tilde{B}'[\vec{\mu}/\vec{u}]\rangle$

8. Cas $A = B|C$.

▷ Supposons $(\nu U)A \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U)(\nu U')B'|C$ avec $B \stackrel{\hat{\alpha}}{\Rightarrow} (\nu U')B'$, $\alpha \neq \tau$ et, $\langle \emptyset, U, [\![B]\!]\rangle \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \mu', \tilde{B}'\rangle$. On a :

a. $\langle \emptyset, U, [\![B|C]\!]\rangle \stackrel{[\hat{\alpha}]}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}' \parallel [\![C]\!]\rangle$

(par hypothèse d'induction et par règle (parallel) des DCC).

b. Comme pour le cas 8.b.

▷ Supposons $(\nu U)A \stackrel{\hat{\tau}}{\Rightarrow} (\nu U)A'$. Alors

Soit ni la règle (communication) ni la règle (fermeture) π-calcul n'interviennent. La transition provient de la réduction d'un agent $\tau.D$. Ce cas se traite comme ci-dessus.

Soit on a :

$B \stackrel{\hat{\bar{u}v}}{\Rightarrow} (\nu U')B'$ et $C \stackrel{\hat{u}v}{\Rightarrow} (\nu U'')C'$, ce dernier provenant de la réduction d'un agent $u(w).D$

Dans ce cas, on a :

a. $\langle \emptyset, U, [\![B]\!]\rangle \stackrel{\widehat{send_{ca}(u,v)}}{\Rightarrow} \langle \emptyset, U \cup U' \cup \vec{\mu}', \tilde{B}'\rangle$

Par réduction de l'agent $[\![\bar{u}v]\!]$ dans $[\![B]\!]$.

Et $\langle \emptyset, U, [\![C]\!]\rangle \overset{\hat{receive_{ca}(u,v)}}{\Rightarrow} \langle \emptyset, U \cup U'' \cup \vec{\mu}'', \tilde{C}'\rangle$
Par réduction de l'agent $[\![u(w).D]\!]$ dans $[\![C]\!]$.
Donc $[\![B]\!]$ possède un agent $send_{ca}(u,v)$ en composition parallèle de premier niveau. Et $[\![C]\!]$ possède un agent $receive_{ca}(u,v).D$ en composition parallèle de premier niveau. La règle (communication) des DCC peut s'appliquer et on a
$\langle \emptyset, U, [\![A]\!]\rangle \overset{\hat{\tau}}{\Rightarrow} \langle \emptyset, U \cup U' \cup U'' \cup \vec{\mu}' \cup \vec{\mu}'', \tilde{B}' \parallel \tilde{C}'\rangle$
b. Pour $\langle \emptyset, L \cup \vec{\mu}, \vec{\mu} = \vec{u} \parallel [\![B]\!][\vec{\mu}/\vec{u}]\rangle$, on effecue le même raisonnement que précédemment. La différence est que $[\![B]\!]$ possède soit un agent $send_{ca}(u,v)$ soit un agent $send_{ca}(\mu_1, \mu_2)$ et dans ce dernier cas, $store(\vec{\mu} = \vec{u} \parallel [\![B]\!][\vec{\mu}/\vec{u}]) \vdash \mu_1 = u \ et \ \mu_2 = v$ (idem pour $[\![C]\!]$).
On conclut comme précédemment.

Soit on a :
$B \overset{\hat{\bar{u}(v)}}{\Rightarrow} B'$ et $C \overset{\hat{uv}}{\Rightarrow} C'$, ce dernier provenant de la réduction d'un agent $u(w).D$
Ce cas se traite comme précédemment, la réduction du π-calcul utilisant la règle (fermeture) à la place de la règle (communication). Mais on a par hypothèse d'induction $v \in U$ □

Lemme A.2 Soit $(\nu U)A$ un agent du π-calcul.
Nous supposons que A ne contient aucun opérateur de restriction.
Si $\langle \emptyset, \emptyset, [\![(\nu U)A]\!]\rangle \overset{[\hat{\alpha_0}]}{\Rightarrow} \langle \emptyset, L_1, \tilde{A}_1\rangle \overset{[\hat{\alpha_1}]}{\Rightarrow} \langle \emptyset, L_2, \tilde{A}_2\rangle$... alors
$(\nu U)A \overset{\hat{\alpha_0}}{\Rightarrow} A_1 \overset{\hat{\alpha_1}}{\Rightarrow} A_2$... avec,
pour tout i, si $\alpha_i = \bar{u}(v)$ alors $v \in L_i$ et,
pour tout i, si $\alpha_i = \bar{u}v$ alors $v \notin L_i$. □

(schéma de) preuve :
On peut effectuer la preuve précédente en inversant les rôles joués par les transitions du π-calcul et des DCC. Le résultat provient du fait que les variables nouvelles introduites dans les transitions DCC sont locales. Cela n'engendre donc pas de transition dont les actions contiennent ces variables.
□

Par les lemmes A.1 et A.2, nous avons une bijection entre les dérivations faibles d'un agent du π-calcul et celles de son encodage. Le résultat de la proposition 7.1 en découle directement.