UNIVERSITÉ PARIS. DIDEROT (Paris 7)

Thèse

pour l'obtention du Diplôme de

Docteur de l'Univerisité Paris. Diderot spécialité Informatique

Fermetures et Modules dans les Langages Concurrents avec Contraintes Fondés sur la Logique Linéaire

présentée et soutenue publiquement par

Rémy Haemmerlé

le 17 janvier 2008

devant le jury composé de

M.	Roberto Di Cosmo	*Examinateur*
M.	Khalil Djelloul	*Examinateur*
M.	François Fages	*Directeur de thèse*
M.	Xavier Leroy	*Examinateur*
Mme	Catuscia Palamidessi	*Rapporteur*
M.	Peter Van Roy	*Rapporteur*

Remerciements

Je remercie en premier lieu mon directeur de thèse, François Fages, qui m'a proposé un sujet passionnant et qui m'a permis de le mener à bien dans les meilleures conditions au sein de son équipe de recherche. Les nombreuses discussions que nous avons pu avoir m'ont été précieuses tout au long de ce travail. Je le remercie aussi de m'avoir donner l'opportunité de travailler sur des thématiques différentes qui ont été autant d'ouvertures enrichissantes. Je suis aussi très reconnaissant du travail de relecture et d'amélioration qu'il a apporté à tous les articles que j'ai pu publier.

Je remercie Catuscia Palamidessi et Peter Van Roy qui ont pris la peine de rapporter ma thèse et qui ont contribué par leurs remarques pertinentes à son amélioration. Je remercie aussi les autres membres du jury : Robert Di Cosmo, Khalil Djelloul et Xavier Leroy. Merci également à tous ceux qui ont participé à la chasse aux fautes et ont ainsi permis de réduire à un niveau acceptable le nombre de coquilles présentes dans mon manuscrit.

Je remercie Sylvain Soliman pour les nombreuses discussions qui m'ont éclairé dans mes recherches aussi bien en Informatique et en Logique qu'en chanson française des années 80. Je remercie aussi Emmanuel Coquery pour son infaillible disponibilité[1] pour discuter de sujets aussi passionnants et variés que le typage, la programmation objet, fonctionnelle ou par contraintes, les systèmes d'expoitation et «Kelvin and Hobbs». Je remercie aussi Daniel Diaz qui m'a permis de mieux comprendre la compilation des langages logiques. Je remercie également les membres de l'équipe PROTHEO du LORIA et les employés d'ILOG que j'ai rencontrés au cours des réunions du projet RNTL MANIFICO.

Je tiens à remercier également toutes les personnes que je n'ai pas encore citées mais que j'ai eu la chance de côtoyer[2] au sein d' l'équipe Contraintes. Je pense notamment à Ludovic Langevine qui m'a fait découvrir le meilleur restaurant d'Uppsala[3], Nathalie Chabrier-Rivier sans qui je ne saurais pas faire du passing torches[4], Aurélie Strobbe qui je l'espère ne regrettera pas son choix, Laurence Calzone qui m'en voudra quoi que j'écrive, Julien Martin qui m'a rapporté une Andechser Hell, Domitille Heitzler qui a toujours la bonne humeur contagieuse et finalement Grégory Batt qui pendant quelques mois, a bien voulu partager son bureau de CR. Je souhaite aussi bon courage aux

[1] notamment pendant la période de rédaction de sa propre thèse

[2] et réciproquement

[3] Le «Tzatziki», Fyristorg 4, UPPSALA, `www.tzatziki.se`

[4] cf. `contraintes.inria.fr/~haemmerl/fire/fire.avi`

nouveaux doctorants de l'équipe, Aurélien Rizk et bien sûr Thierry Martinez qui aura la lourde charge de me remplacer comme moteur du projet SiLCC.

Je remercie également Nadia Mesrar, Sylvie Loubressac et Emmanuelle Grousset sans qui les quelques années passées à l'INRIA auraient été, me semble-t-il, un cauchemar administratif. J'ai aussi une pensée pour les membres de l'équipe Moscova et les participants au défunt club jonglage.

Je remercie finalement les membres de ma famille et tous mes amis non-informaticiens qui se sont intéressés[5], durant ces quatre années, à mes travaux de recherche.

[5]ou qui ont admirablement fait semblant

Résumé

Les langages concurrents avec contraintes fondés sur la logique linéaire (LCC) permettent de modéliser une grande variété de mécanismes de la programmation par contraintes allant des coroutines de Prolog à la de propagation de contraintes sur les domaine finis en passant pas les opérations de changement d'états dans les algorithmes de contraintes globales. Le projet SiLCC («*SiLCC is Linear logic Concurrent Constraint programming*») a pour objectif la conception et l'implémentation d'un langage riche combinant contraintes, concurrence et changement d'états suivant une architecture modulaire. Le système implémentant ce langage devra être complètement bootstrappé au dessus d'un langage LCC noyau et fourni avec un grand nombre de bibliothèques. Les travaux menés dans cette thèse ont pour objectif la conception d'un système de modules pour les langages LCC, prérequis au développement effectif du système SiLCC.

Traditionnellement les systèmes de modules pour les langages de programmation sont vus comme indépendants du langage sous-jacent. Cette approche a cependant plusieurs inconvénients. De tels systèmes interfèrent souvent avec les constructions natives du langage, sont en partie redondants avec certains mécanismes du langage hôte et font perdre les résultats de sémantique formelle qui peuvent ne plus s'appliquer aux programmes modulaires. Cette thèse étudie l'internalisation dans les langages LCC d'un système de modules sans modifier la logique sous-jacente. Il en résulte un langage opérationnellement très simple et mono-paradigme qui incorpore nativement un grand nombre de concepts de programmation tels que le non determinisme, les contraintes, l'affectation impérative, la gestion de la concurrence, les fermetures et l'encapsulation de code. Cette grande simplicité permet alors d'obtenir de façon directe une sémantique logique premier ordre.

Dans la première partie de cette thèse nous montrons que les déclarations de procédures des langages LCC peuvent être supprimées au profit d'un nouveau type d'agent *asks* persistants tout en préservant la sémantique logique du langage. Nous montrons que le langage qui résulte de cette internalisation des déclarations fournit toute la puissance d'expression des fermetures des langages fonctionnels et des calculs de processus classiques à travers des encodage où les variables jouent le rôle de canaux de communication dynamiques. Dans la seconde partie nous définissons un système de modules pour cette classe de langages, dans lequel les modules peuvent être référencés par des variables. Nous démontrons que grâce à ce système de modules le masquage de code s'obtient alors par l'opérateur usuel de masquage des variables. Puis nous présentons un prototype d'implémentation d'une instance de ce système de modules et illustrons son utilisation en programmation avec contraintes.

Mots clés : programmation concurrents avec contraintes, modules, fermetures, logique linéaire.

Abstract

The Linear logic Concurrent Constraint languages (LCC) allow to model a large number of constraint programming systems, from Prolog coroutines to constraint propagation mechanisms such as those implemented in the finite domain solvers and also state change as in global constraint algorithms. The main objective of the SiLCC project ("SILCC is Linear logic Concurrent Constraint programming") is to define a rich language combining constraints, concurrence and state change following a modular architecture. The system implementing this language should be completely bootstrapped over a kernel LCC language provided with a large number of libraries. The main topic of this thesis is the conception of a module system for LCC, prerequisite of the effective development of the SiLCC system.

Module systems for logic languages with constraints are usually viewed as independent from the underlying language. This approach has however some drawbacks. First such module systems often interfere with the programming construct, then they are in part redundant with scope mechanisms of the host language and result in LCC languages losing the results of logical semantics that cannot anymore applied to modular programs. This thesis studies the internalization in of a module system without modifying the underlying logic. The languages resulting from this internalization have simple mono-paradigm semantics and embed numerous programming concepts such that non determinism, constraints, imperative assignment, concurrence, closures et code encapsulation. This great simplicity of this approach allows to obtain directly a first-order logical semantics.

In the first part of this thesis, we show that the declarations usual in LCC languages can be taken out, to the benefit of a new "persistent ask" agent, while preserving the logical semantics of the language. We also show that the languages resulting from this internalization have all the expressive power of functional languages and usual process calculi. In the second part we define a module system for this class of languages, in which modules can be referenced by variables. We show then than the implementation hiding can be obtained with the usual variable hiding operator. Finally we present a prototype of implementation of one instance of this module system and illustrate its use in logical constraint programming.

Keywords: concurrent constraint programming, modules, closures, liner logic.

Table des matières

Chapitre 1

Introduction

Une histoire de la programmation logique

La programmation logique est un paradigme de programmation introduit par A. Colmerauer et R. Kowalski [Kow74] au début des années 70, dans lequel les programmes sont représentés par des clauses logiques et exécutés par un principe de déduction automatique. Plus tard, D. Warren [War83] montra que le langage Prolog, instance de ce paradigme, pouvait être compilé de façon efficace sur des ordinateurs classiques ; pour cela il définit une machine abstraite, connue aujourd'hui sous le nom de *Warren's Abstract Machine* (ou WAM). Néanmoins Prolog souffrait de quelques inconvénients, notamment l'impossibilité de manipuler des structures de données autres que des termes, de même que celle de diriger les calculs par les données et non par les buts.

La programmation logique par contraintes (notée PLC) définie par J. Jaffar et J.L. Lassez [JL87], contribua à résoudre ces défauts. En effet il est possible de généraliser la théorie et les techniques de compilation de la programmation logique sur n'importe quelle structure mathématique (ou domaine de discours) définissable par une théorie décidable et où les contraintes sont résolues de façon concurrente au mécanisme de résolution des buts Prolog. Ainsi Prolog peut être vu comme un langage PLC où le domaine de discours se réduit à l'algèbre des termes du premier ordre, le domaine de Herbrand et la résolution de contraintes à l'unification.

Malgré tout, PLC souffre toujours de quelques défauts : le manque de contrôle dans l'exécution d'un programme, l'absence d'ordre supérieur et l'impossibilité de manipuler des données impératives, contraignent la liberté du programmeur. Ainsi les principaux interpréteurs/compilateurs de programmes logiques par contraintes (citons par exemple GNU-Prolog [Dia03] et SICStus Prolog [Swe04]) mettent à la disposition du programmeur un grand

nombre de mécanismes extra-logiques, permettant entre autres un meilleur contrôle, la manipulation de données impérative et même la possibilité de définir de nouveaux solveurs de contraintes.

Les langages Concurrents avec Contraintes (CC), introduits par Maher [Mah87] et Saraswat [Sar93b], généralisent la programmation logique avec contraintes par l'ajout d'une primitive de synchronisation fondée sur l'implication de contraintes. Ils définissent ainsi un calcul concurrent où des *agents* peuvent communiquer et se synchroniser par l'intermédiaire d'un tableau noir commun, appelé *store*, qui est une contrainte représentant l'accumulation d'information partielle sur les variables. Un agent peut donc ajouter une information en postant dans le store une contrainte – comme le ferait un but PLC – ou attendre que le store implique une contrainte. En ajoutant le non-déterminisme on obtient un langage relativement puissant permettant la reconstruction de certains traits des systèmes de programmation par contraintes cachés dans le schéma PLC, comme, par exemple, la gestion des coroutines et le mécanisme de propagation de contraintes.

Les langages concurrents avec contraintes basés sur la logique linéaire (LCC), introduits informellement dans [Sar93a] puis développé dans [Rue97, Sch99, FRS01, Sol01] par l'étude de la sémantique logique des langages CC, généralisent les langages CC en considérant les contraintes comme des formules de la logique linéaire de Girard [Gir87]. Comme l'a montré [Sch99] ces langages de contraintes permettent un contrôle accru de la concurrence mais aussi la modélisation naturelle des traits impératifs, des variables attribuées et plus généralement des opérations de changements d'états.

Un noyau pour la programmation logique avec contraintes

Traditionnellement, les systèmes de programmation logique avec contraintes s'appuient sur un solveur de contraintes figé écrit dans un langage impératif – typiquement C – lié à un interpréteur de type WAM chargé de la résolution logique. Dans les systèmes suivant cette approche – on peut citer par exemple Prolog III [Col87], PLC(R) [HMSY87] et CHIP [DSH90] – la séparation entre le langage logique et le solveur de contraintes utilisé comme une «boîte noire» rend problématique l'analyse, la combinaison et l'extension de solveurs de contraintes.

Plusieurs solutions ont été proposées pour permettre plus de flexibilité. D'un coté il y a l'approche « bibliothèque » qui consiste à utiliser les mé-

canismes de coroutines et d'assignations pour implanter, comme des bibliothèques, des solveurs de contraintes [CFS93, Hol95]. Il y a aussi l'approche dite de « la boîte de verre », utilisé dans cc(FD) [HSD98] et GNU Prolog [Dia95, DC01], qui consiste à réduire la granularité des primitives en se contentant d'un jeu réduit de contraintes primitives (les indexicaux) allié à des mécanismes de propagation de type CC (contraintes réifiées). Ces démarches présentent toutefois quelques limites. D'une part la sémantique des mécanismes primitifs mise en jeu ne s'exprime pas dans la sémantique du langage hôte. De plus, les contraintes « globales » généralement spécifiées grâce à des propriétés de graphes [BCDP05] sont difficilement reconstructibles à partir de contraintes élémentaires.

Il semble alors raisonnable de définir un langage noyau, le plus petit possible et suffisamment expressif pour reconstruire dans un cadre unifié de programmation logique avec contraintes incluant différents solveurs et des bibliothèques génériques. Schächter [Sch99] a démontré que la classe des langages LCC offrait un cadre logiquement bien fondé incluant non-déterminisme, concurrence et changement d'état, idéal pour la programmation de solveurs de contraintes complexes.

De ces considérations est né le projet SiLCC (pour «*SiLCC is Linear logic Concurrent Constraints programming*») [Fag00, Fag04, Hae05]. Ce projet a pour objectif la conception d'un système LCC modulaire basé sur un système de contrainte noyau. Afin de pouvoir être modifié et étendu par le programmeur lui-même, le système implémentant ce langage se doit d'être complètement *bootstrappé* au dessus d'un noyau le plus petit possible et fournit avec un grand nombre de bibliothèques à usage générique. De ce point de vue, SiLCC à des préoccupations proches de Oz [MMR95], mais dans un cadre non pas multi-paradigme mais mono-paradigme. Nous avons identifié que la définition formelle d'un système de modules à la fois simple, puissant et implémentable était la première étape du développement du système SiLCC [Hae05].

Un système de modules internalisé pour la programmation logique avec contraintes

La modularité est un trait essentiel des langages de programmation modernes pour la conception d'applications de grande taille. En effet elle autorise la factorisation du développement de logiciels en permettant la réutilisation de code existant et l'élaboration de bibliothèques génériques. De plus elle

facilite le développement coopératif car la segmentation qu'elle impose améliore la lisibilité du code. Le point de départ des travaux présentés dans ce mémoire est donc la volonté de fournir au cadre SiLLC un mécanisme de module expressif.

Généralement les systèmes de modules pour les langage de programmation sont vus comme indépendants du langages sous-jacent. Cette approche défendue par Leroy [Ler00] a néanmoins quelques inconvénients. En effet de tels systèmes peuvent interférer avec les constructions natives du langage. On peut citer le cas des méta-appels Prolog qui se généralisent très mal aux cadres modulaires syntaxiques, fonctionnels ou objets[1]. De plus ces systèmes reproduisent bien souvent des mécanismes du langage hôte, comme par exemple les mécanismes de portée (masquage du code et portée des variables) ou les mécanismes de paramétrisations (foncteurs et fonctions d'ordre supérieur). Finalement l'association d'un système de modules et d'un langage hôte basé sur des paradigmes différents conduit souvent à l'abandon des résultats de sémantique formelle existants. Par exemple, il n'est pas possible de définir simplement un sémantique logique à un langage logique utilisant un système de modules fonctionnels ou objets.

La construction d'un système de modules à partir des constructions natives du langage LCC semble être un moyen simple de résoudre ces problèmes. Cette internalisation du système de module a en plus l'avantage de cadrer avec notre objectif de minimisation des concepts de notre langage. De façon générale, cette opération est réalisable aisément dans un grand nombre de langages[2]. Cependant, afin d'obtenir un système de modules expressif permettant l'encapsulation et la paramétrisation de code, le langage noyau doit intégrer des mécanises similaires. De ce point de vue le langage LCC dispose déjà d'un mécanisme de gestion portée, à travers le quantificateur $\exists$, mais ne possèdent aucun mécanisme de paramétrisation. Partant de ce constat, la première partie du travail présenté ici a consisté à ajouter un mécanisme de fermeture, c.-à-d. la possibilité de manipuler comme objet de première classe du code associé à un environnement, aux langages LCC, la seconde partie étudiant l'internalisation du système de modules dans le langage obtenu.

[1] Nous en reparlerons au chapitre 6.

[2] L'annexe A illustrent comment cela peut être fait dans un langage fonctionnel et un langage logique.

Un plan de la thèse

Cette thèse démontre qu'il est possible d'internaliser dans les langages à variables logiques comme LCC un système de module simple et puissant. Cette démonstration est divisée en deux parties. La première partie s'intéressent à l'internalisation des mécanismes de fermetures dans les agents LCC. La seconde partie étudie, quant à elle, l'internalisation des modules dans le langage résultant.

Le chapitre 2 introduit brièvement les langages ainsi que les principaux traits extra-logiques des systèmes PLC les plus couramment utilisés. Le chapitre 3 présente les langages LCC dans un cadre étendu où les déclarations sont abandonnées au profit d'un nouveau type d'agent *ask* persistant. Il y est aussi démontré que les résultats de sémantique logique de [FRS01] sont préservés malgré ce nouveau type d'agents et que les traits extra-logiques de PLC trouvent dans LCC un cadre logiquement bien fondé. Le chapitre 4 montre que les langages ainsi obtenus possèdent toute la puissance expressive des fermetures dans langages fonctionnels.

La deuxième partie commence au chapitre 6, par une étude préliminaire sur la modularité dans le contexte restreint des langages PLC. Dans le chapitre 7 les modules sont ajoutés aux langages LCC, comme une simple couche de sucre syntaxique restreignant l'ensemble des agents définissables. Le chapitre 8 reprend, ensuite, les résultats des chapitres 4 et 5 dans le cadre modulaire. Le chapitre 9 présente une implémentation prototype du système de modules.

Enfin pour conclure nous exposons brièvement les perspectives qu'ont ouverts ces travaux.

Première partie

Vers les Langages Concurrents avec Contraintes fondé sur la Logiques Linéaire

Chapitre 2

Programmation Logique avec Contraintes : Théorie et Pratique

Sommaire

2.1 Introduction

La *Programmation Logiques avec Contraintes* (PLC) [JL87] a été introduite par J. Jaffar et J.L. Lassez en 1986. Bien qu'elle permette de résoudre élégamment et efficacement de nombreuses difficultés de programmation en

Prolog pur – voir à titre d'exemple [HMS92, CDKM95] – les langages PLC pures manquent encore d'expressivité pour pouvoir être considérés comme des langages généralistes. Ainsi les principaux interpréteurs/compilateurs PLC mettent à la disposition du programmeur de nombreuses extensions extra-logiques permettant, entre autres, un meilleur contrôle, la manipulation de données impératives et même la possibilité de définir de nouveaux solveurs de contraintes dans le même langage. Bien que ces différents mécanismes fassent aujourd'hui partie du folklore de la programmation par contraintes et qu'ils sont implémentés dans des extensions de la machine abstraite de Warren [War83, AK91], ils sortent plus ou moins du cadre théorique de la PLC.

Ce chapitre rappelle le cadre théorique des langages PLC et présente les principaux traits extra-logiques des systèmes de programmation PLC les plus couramment utilisés. Les notations utilisées proviennent en grande partie du livre [Fag96].

2.2 La Programmation Logique par Contraintes

2.2.1 Système de contraintes

Définition 2.1 (Langage des contraintes)**.** Soit le langage du premier ordre défini par :

- un ensemble dénombrable $\mathcal{V}$ de variables,
- un ensemble Σ_F de symboles de fonction,
- un ensemble Σ_C de symboles de relation, appelé ensemble de *symboles de contraintes*, contenant *true* et $=$.

Une *contrainte atomique* est une proposition de ce langage. Le *langage des contraintes*, noté $\mathcal{C}$, est une partie des formules logiques, contenant les contraintes atomiques et supposée close par conjonction et renommage des contraintes atomiques. Les éléments de ce langage (appelés simplement *contraintes*) seront notés $c, d, \ldots$.

Dans la suite, $x, y, \ldots$ dénoteront des éléments d'un ensemble dénombrable $\mathcal{V}$ de variables et $\vec{x}$ une séquence finie de variables de $\mathcal{V}$. L'ensemble des variables libres d'une formule F sera noté $\mathcal{V}(F)$.

Définition 2.2 (Système de contraintes)**.** Un *système de contraintes* est une paire $(\mathcal{C}, \mathcal{T})$ où $\mathcal{C}$ est un langage de contraintes et $\mathcal{T}$ une théorie axiomatique sur le vocabulaire des contraintes.

2.2.2 Les programmes PLC($\mathcal{X}$)

Dans ce chapitre, un système de contraintes $\mathcal{X} = (\mathcal{C}, \mathcal{T})$ sera supposé donné. De même, un **atome** désignera une proposition du langage formée uniquement de Σ_P, Σ_F et V.

Définition 2.3 (Programme PLC($\mathcal{X}$))**.** $A :\!- c \mid \alpha$. Une *clause PLC($\mathcal{X}$)* est une formule logique de la forme :

$$\forall\, (A \vee \neg c_1 \vee \ldots \neg c_m \vee \neg A_1 \ldots \neg A_n)$$

où les $c_i \in \mathcal{C}$ sont des contraintes atomiques et les A_i des atomes. Cette formule est notée plus simplement

$$A :\!- c_1, \ldots, c_m, A_1, \ldots A_n. \quad \text{ou de façon condensée} \quad A :\!- c|\alpha.$$

où c est la conjonction des contraintes de la clause et α dénote la séquence d'atomes $A_1, \ldots, A_n$. Un *programme PLC($\mathcal{X}$)* est un ensemble fini de clauses PLC($\mathcal{X}$).

Définition 2.4 (But PLC($\mathcal{X}$))**.** Un *but PLC($\mathcal{X}$)* est une formule logique de la forme :

$$\neg c_1 \vee \ldots \neg c_m \vee \neg A_1 \ldots \neg A_n$$

où les c_i et les A_i respectent les mêmes conditions que dans la définition précédente. On le note plus simplement

$$c_1, \ldots, c_m, A_1, \ldots A_n. \quad \text{ou de façon condensée} \quad c|\alpha.$$

où c est la conjonction des contraintes de la clause et α dénote la séquence d'atomes $A_1, \ldots, A_n$. $\Box$ dénote la séquence d'atomes vide.

2.2.3 Sémantique opérationnelle

La sémantique opérationnelle d'un programme PLC est définie de façon très simple à l'aide d'une unique règle de déduction, appelée CSLD (pour résolution Linéaire pour les programmes Définis avec Contraintes et Sélection du littéral).

Définition 2.5 (Relation de Transition)**.** Soit $(\mathcal{C}, \mathcal{T})$ un système de contraintes et $\mathcal{P}$ un programme PLC sur ce système. La relation de transition $\longrightarrow$ sur les buts est définie comme étant la plus petite relation satisfaisant le principe de la résolution CSLD :

$$\frac{(p(\vec{x}) :\!- d|\beta.)\theta \in \mathcal{P} \qquad \mathcal{T} \vdash \exists(c, \vec{x} = \vec{y}, d)}{(c|\alpha, p(\vec{y}), \alpha'.) \longrightarrow (c, \vec{x} = \vec{y}, d|\alpha, \beta, \alpha'.)}$$

où θ est un renommage à l'aide de variables fraîches.

Définition 2.6 (Sémantique Opérationnelle)**.** Une *dérivation CSLD* pour un but B_0 est une suite finie ou infinie de buts (B_i) telle que $B_j \longrightarrow B_{j+1}$ (i étant positif). Un *dérivation succès* est une dérivation CSLD finie dont le dernier élément B_n est un but ne contenant que des contraintes, c.-à-d. , $B_n = c|\Box$. $\exists\vec{x}(c)$ où $\vec{x} = fv(c) \setminus fv(B_0)$ est alors appelé *réponse calculée.*

La proposition suivantes formule le fait que le long d'une dérivation le mécanisme de résolution ne fait qu'ajouter des contraintes.

Proposition 2.7 (Extensivité des contraintes)**.** *Soient $c|\alpha$ et $d|\beta$ deux buts PLC($\mathcal{S}$).*

$$Si \quad c|\alpha \xrightarrow{*} d|\beta \quad alors \quad \mathcal{T} \vdash d \supset c.$$

Définition 2.8 (Stratégie de Sélection des Atomes [JMMS98])**.** Une *stratégie de sélection des Atomes* est une fonction qui pour toute dérivation retourne un atome dans le dernier but de cette dérivation.

Une dérivation est *générée* par une stratégie de sélection S, si tout les choix de sélection des atomes dans la dérivation ont été faits suivant S, c.-à-d. si la dérivation est de la forme

$$B_0 \longrightarrow B_1 \longrightarrow \ldots \longrightarrow B_n \longrightarrow \ldots$$

alors pour chaque $i \leq 0$ l'atome sélectionné dans l'état B_i est

$$S(B_0 \longrightarrow B_1 \longrightarrow \ldots \longrightarrow B_i).$$

Un but B_0 est *accessible* pour un but B_n via une stratégie S, s'il existe une dérivation $B_0 \xrightarrow{*} B_n$ générée par S.

Proposition 2.9 (Indépendance vis-à-vis de la Stratégie de Sélection des Atomes [JMMS98])**.** *Soient S et S' deux* stratégies de sélection des Atomes. *Si c est une réponse calculée du but G via la stratégie S, alors il existe une réponse calculée c' pour G via la stratégie S' telle que $\mathcal{T} \models c \leftrightarrow c'$.*

2.2.4 Sémantique logique

Le sens logique d'un programme PLC permet de s'intéresser aux conséquences logiques d'un programme et de la théorie du système de contraintes. La question est de savoir si une certaine contrainte c implique le but G :

$$\mathcal{P}, \mathcal{T} \vdash c \supset G$$

Théorème 2.10 (Correction de la résolution CSLD)**.** [JL87]

$$Si\ c\ est\ la\ réponse\ calculée\ d'un\ but\ G\ alors\ \ \mathcal{P}, \mathcal{T} \vdash c \supset G.$$

Théorème 2.11 (Complétude de la résolution CSLD par rapport à la théorie de la structure)**.** [Mah87] *Si* $\mathcal{P}, \mathcal{T} \vdash c \supset G$ *alors il existe un ensemble fini* $\{c_1, \ldots, c_n\}$ *de réponses calculées pour le but* G *tel que :* $\mathcal{T} \vdash (c \supset c_1 \vee \cdots \vee c_n)$.

2.2.5 Exemple de programme PLC

Le programme PLC suivant est un exemple standard distribué avec l'implémentation CLP(R) de l'université de Monash [JMSY92]. Il définit la formule des intérêts d'un emprunt par l'intermédiaire de l'appel au prédicat `mortage(P,T,I,M)` où `P` est le capital, `T` la durée en mois, `I` le taux et `M` le montant mensuel. Le programme est capable de calculer des réponses partiellement instanciées pour différentes combinaisons d'entrées.

Exemple 2.12 (Mortage). :

```
mortgage(P,T,I,M):- T=1, M = P*(1+I).
mortgage(P,T,I,M):- T>1, mortgage(P*(1+I)-M , T-1, I, M).


?- mortgage(120000, 120, 0.01, M).

M = 1721.6513808 ?

yes

?- mortgage(P, 120, 0.01, 1721.6513808).

P = 120000 ?

yes


?- mortgage(P, 120, 0.01, M).

P=69.700522*M ?

yes
```

2.3 Traits Extra-logiques

Les notions examinées dans cette section sortent du cadre théorique de la programmation logique avec contraintes. Néanmoins elles ont prouvé leur

utilité en termes de facilité de programmation et d'efficacité. Aujourd'hui, aucun utilisateur de systèmes PLC ne saurait programmer sans elles. On peut retrouver dans [CR93] l'histoire de quelques unes d'entre elles. Nous montrerons dans le chapitre suivant que toutes ces impuretés trouvent dans les langages LCC, un cadre mono-paradigme formelle très simple et logiquement bien fondé.

2.3.1 Les termes *mutables*

L'extensivité (proposition 2.7) d'un système de contraintes interdit la manipulation de données impératives dans un programme PLC. En effet une fois instanciée, une variable ne pourra plus évoluer dans le temps. Il existe cependant de nombreux algorithmes où l'affectation destructive est essentielle pour les performances.

Pour simuler les changements d'états (c.à.d. l'opération d'affectation des langages impératifs) le système SICStus Prolog [Swe04] fournit au programmeur les *termes mutables*. Un terme mutable représente une cellule mémoire dont le contenu est un terme qui peut être modifié de façon arbitraire puis restauré au *backtracking*. Au top-level un terme mutable de valeur courante `X` sera représenté par le terme `'$mutable'(X)`. Malgré son nom, le terme mutable n'est pas un terme à proprement parler ; notamment il ne peut être unifié qu'à des variables libres.

Il est possible de créer un terme mutable `Mut` par l'exécution du but `create_mutable(Datum, Mut)` ; ce dernier crée une nouvelle cellule mémoire dans laquelle est initialement stockée la valeur `Datum`. Il est possible d'unifier la valeur d'un *terme mutable* `Mut` au terme `Datum` à l'aide de l'appel `get_mutable(Datum, Mut)`. Enfin, on peut modifier le contenu du *terme mutable* et y stocker la valeur `Datum` par l'appel du but `update_mutable(Datum, Mut)`.

Exemple 2.13. :

```
?- create_mutable(2, Mut).

Mut = '$mutable'(2)

yes

?- create_mutable(2, Mut),
   get_mutable(Datum, Mut),
   update_mutable(3, Mut).
```

```
Datum = 2,
Mut = '$mutable'(3)

yes
```

2.3.2 Les variables globales

Le comportement d'un programme informatique réel est souvent dicté par de nombreux paramètres. Or Prolog ne permet pas de manipuler d'information de façon globale. Ainsi, si tous les prédicats d'un programme sont susceptibles d'accéder à ce type d'information, il est nécessaire de passer en argument de tous ces prédicats l'ensemble des données paramétrant le programme. Néanmoins cette méthode de programmation devient vite pénible à mettre en œuvre et produit du code relativement mal compilé par la WAM.

Pour contourner ces problèmes, le système GNU-Prolog [Dia03], propose d'utiliser des *variables globales*. Ce mécanisme permet d'associer à une clé (une constante de Σ_F), un terme quelconque, ce dernier pouvant être récupéré par l'intermédiaire de sa clé. Tout comme les *termes mutables* l'affectation d'un terme à une clé est destructive, ainsi les *termes mutables* peuvent être vu comme une version locale de variable globale. L'affectation d'un terme `Value` à une clé `GVarName` ce fait par l'intermédiaire de l'appel `g_link(GVarName, Value)`. Il est possible d'unifier la valeur associé à une clé `GVarName` au terme `Value` en exécutant `g_read(GVarName, Value)`.

Exemple 2.14. :

```
?- g_link(foo, 2),
   g_read(foo, X).

X = 2

yes

?- g_link(foo, 2),
   g_read(foo, X),
   g_link(foo, 3),
   g_read(foo, Y).

X = 2,
Y = 3
```

```
yes
```

2.3.3 Les méta-appels

Pour compenser le manque d'ordre supérieur du langage théorique – c.-à-d. la possibilité de manipuler du code comme un objet de première classe – Prolog a rapidement incorporé la notion de *méta-appel*. Ce mécanisme ISO [Int95] permet d'interpréter, comme des buts, des termes créés dynamiquement.

Cette primitive est extra-logique dans le sens où les termes et les prédicats ne sont pas définis dans le même univers, en effet, théoriquement, ils ne sont pas construits à partir du même alphabet de symboles. Néanmoins, dans la pratique, les termes et les prédicats étant construits sur le même alphabet[1] la correspondance entre termes et prédicats est naturelle.

En programmation, l'une des utilisations les plus standard de l'ordre supérieur est la définition d'itérateur sur des structures de données. On peut citer comme exemple canonique la fonction $map : (A \rightarrow B) \rightarrow list(A) \rightarrow list(B)$ qui applique une fonction passée en premier argument, à tous les éléments d'une liste passée en second argument.

De façon analogue, il est possible, à l'aide du prédicat `call/2`, de définir deux prédicats binaires `forall/2` et `exists/2` qui vérifient que chaque élément d'une liste (resp. au moins un) vérifie une propriété passée en second argument. On supposera, pour cet exemple, que l'on dispose du prédicat ISO `=..` qui permet de construire un terme à partir d'une liste contenant le symbole de fonction suivant des arguments du terme.

Exemple 2.15 (Itérateurs).

```
forall([], _).
forall([H|T], P):- G=..[P|H], call(G), forall(T, P).

exists([H|_], P):- G=..[P|H], call(G).
exists([_|T], P):- exists(T, P).
```

2.3.4 Les assertions dynamiques de clauses

En poussant encore plus loin le concept de méta-donnée – c.-à-d. la manipulation de code comme des objets de première classe – permise par l'in-

[1] Nous verrons dans la deuxième partie que cela n'est pas toujours le cas dans un système modulaire

troduction des meta-appels, Prolog permet aussi la manipulation de clauses à travers les mécanismes ISO d'*assertion*.

D'un point de vue pratique, un programme PLC est vu comme une succession de clauses, appelée base de règles, l'ordre dans lequel sont classées les clauses étant utilisé pour déterminer l'ordre d'exécution des clauses. Les prédicats d'assertion `asserta/1` et `assertz/1` permettent au programmeur Prolog d'ajouter respectivement au début ou à la fin de la base de règles la clause correspondante au terme passé en argument.

Ces traits sont particulièrement utilisés pour stocker de l'information, de façon globale et non backtracante. L'exemple suivant montre une autre utilisation élégante de la modification dynamique de programme.

Exemple 2.16 (Mémoisation). L'implémentation récursive naïve de la relation de définition de la suite de Fibonacci est un algorithme connu pour sa grande inefficacité. En effet celui-ci calcule de façon répétée, les mêmes valeurs. Une façon élégante d'améliorer significativement le comportement d'un tel algorithme est de sauvegarder les valeurs intermédiaires par *mémoïsation*. La complexité de calcul, exponentiel dans le cas naïf, devient alors linéaire.

Comme le montre l'exemple suivant l'utilisation des assertions de clauses rend cette technique très naturelle dans les systèmes Prolog pourvus de ce mécanisme. L'idée est d'utiliser l'implémentation récursive naïve, mais d'asserer à la fin d'un calcul du $\texttt{N}^{ième}$ nombre Fibonacci, le fait $fib(N, F)$. Ainsi lors du prochain appel à $fib/2$ pour le calcul du $\texttt{N}^{ième}$, le second argument sera unifié directement à la valeur F précédemment calculée.

```
fib(0, 0).
fib(0, 0).
fib(N, F) :- N > 1,
             fib(N-1, F1), fib(N-2, F2), F=F1+F2,
             asserta(fib(N, F)).
```

2.3.5 Les coroutines

Introduite dans le système Prolog II par A. Colmerauer [Col82] , la première *coroutine* Prolog, définie par la primitive `geler/2` (appelée `freeze/2` dans tous les systèmes modernes), permet de retarder l'exécution d'un but sur l'instanciation d'une variable : `freeze(Var, Goal)` réveille le but `Goal` dès que `Var` est instanciée. Parce qu'elle permet un contrôle sur l'ordre d'exécution des buts dirigé par les données, cette primitive rend possible une résolution plus efficace de certains problèmes.

Exemple 2.17. Les deux buts suivants montrent que le but `write(X='X')` suspendu à l'aide du `freeze/2` n'est exécuté que si `X` est un instanciée.

```
?- freeze(X,write('X'=X)).

freeze(X, write(X='X'))

yes

?- freeze(X,write(X='X')), X=1.

1=X

X -1
yes
```

Exemple 2.18 (Système de contraintes naïf). En interprétant un but gelé comme une contrainte, le `freeze` permet de disposer d'une implémentation naïve (correcte mais non complète) d'un système de contrainte. Par exemple, dans le but suivant, on peut interpréter le but `freeze(X,X>2)` comme une contrainte sur `X` qui ne sera évaluée que quand `X` sera instancié – l'implémentation du prédicat ISO `>` ne pouvant être exécutée que sur des termes instanciés :

```
?- freeze(X,X>2), membre(X,[0,1,2,3,4]).

 X=3 ?;
 X=4 ?;
 no.
```

2.3.6 Les variables attribuées

Pour implémenter des algorithmes de façon efficace, il est souvent intéressant d'étendre les structures de donnée du langage PLC. Pour cela il faut être capable d'étendre la sémantique de la contrainte d'égalité sur un nouveau domaine de contraintes. Il est donc nécessaire de modifier l'algorithme d'unification

Comme l'exemple précédent le montre, les coroutines permettent de modifier le comportement de l'unification en provoquant un échec si la contrainte égalité qu'elle traduit n'est pas cohérente avec le but geler. Néanmoins un mécanisme tel que le `freeze` ne permet en aucun cas de tirer parti de l'unification de deux variables libres ; pour cela un mécanisme plus complexe doit être introduit.

Les variables *attribuées*, introduite par Holzbaur [Hol92] sont des variables Prolog spéciales, qui peuvent être associées, par l'intermédiaire d'une pri-

mitive, à un terme appelé *attribut*. Toutes les primitives Prolog classiques observent les variables attribuées comme des variables ordinaires. Par contre l'unification d'une telle variable avec un terme instancié ou une autre variable attribuée est redéfinie au sens d'un prédicat logique défini par l'utilisateur lui-même.

Il est possible d'attacher un attribut `Att` à une variable `X` par l'appel `put_att(X, Att)`. Ce mécanisme est destructif (mais reste transparent pour le backtracking), c.-à-d. qu'un nouvel attribut peut-être attaché à une variable déjà attribuée, auquel cas l'ancien est perdu. L'appel `get_att(X, Att)` unifie `Att` à l'attribut courant de `X`.

Si l'utilisateur manipule des variables attribuées, il doit définir le prédicat `verify_attributs/3`. Ainsi, `verifify_attributs(Var, Value, Goal)` sera appelé par le système, à chaque fois que l'algorithme d'unification tentera de lier une variable attribuée `Var` à un terme instancié ou une autre variable attribuée `Value`. Si cet appel réussit, `Var` est alors liée à `Value` sinon l'unification échoue. Par convention `Goal` doit être unifié par le prédicat `verifify_attributs/3` à un but qui est appelé après l'unification effective de `Var` à `Value`.

Ce mécanisme est très puissant, en effet il permet de redéfinir deux des notions présentées précédemment. Pour preuve, un programme implémentant la coroutine `freeze` (exemple 2.19) et une implémentation des *termes mutables* (exemple 2.20) sont proposés ci-dessous.

Exemple 2.19 (Implémentation Simplifiée de la Coroutine `freeze`).

```
freeze(X, Goal):-
    get_att(X, Goal_Old), !,
    put_att(X, (Goal_Old, Goal)).
freeze(X, Goal):-
    put_att(X, Goal).

verify_attributes(X, Y, G):-
    nonvar(Y), get_att(X, G).
verify_attributes(X, Y, true):-
    get_att(X, Goal_X),
    get_att(Y, Goal_Y),
    put_att(Y, (Goal_X, Goal_Y)).
```

Exemple 2.20 (Implémentation Simplifiée des termes *mutable*).

```
create_mutable(Datum, Mut):-
    put_att(Mut, Datum).
```

```
get_mutable(Datum, Mut):-
    get_att(Mut, Datum).

update_mutable(Datum, Mut):-
    put_att(Mut, Datum).

verify_attributes(_,_,_):-fail.
```

2.4 Discussion

Les langages PLC permettent de résoudre élégamment et efficacement un grand nombre de problèmes. En outre cette classe de langage admet une sémantique simple accompagnée de nombreuses techniques d'analyse de programmes – seule la sémantique logique a été présentée ici. Néanmoins, comme le montrent les nombreuses extensions non-logiques dont disposent les systèmes PLC courants, aucune instance pure de PLC ne peut être considérée comme un langage généraliste, il lui manque au moins un mécanisme d'assignation impérative et la gestion native de la concurrence. Comme l'a montré Schächter [Sch99], les langages concurrents avec contraintes fondés sur la logique linéaire (LCC) forment une classe de langages logiquement bien fondée qui fournit en plus de toute la puissance expressive de PLC, ces deux traits fondamentaux.

Cependant, il manque toujours à ces langages un système de modules permettant le développement de bibliothèques, comme cela peut-être fait, par exemple, en Java. Afin de paramétrer dynamiquement le comportement de ces bibliothèques génériques (comme les itérateurs de liste de l'exemple 2.15) le langage doit posséder une notion de fermeture, c'est à dire la possibilité de manipuler du code avec un environnent partiel comme un objet de premier classe.

Warren [War82] a montré que les mécanismes de méta-appels présentés en sous-section 2.3.3 possédaient la puissance expressive des fermetures. Néanmoins l'utilisation du `call`, à la place des fermetures, n'est pas entièrement satisfaisante. Tout d'abord, comme cela a été dit plus haut, un tel mécanisme ne possède pas naturellement de sémantique logique. De plus, l'objet passé comme méta-donnée n'est pas atomique (c'est un terme qui peut être décomposé et analysé à loisir) et n'est donc pas vue comme une boîte noire par le code qui le manipule, comme cela devrait être le cas avec une fermeture. Finalement comme le montrera le chapitre 6, cette primitive pose des problèmes aux systèmes de modules.

Dans la suite de ce mémoire nous montrons comment l'ajout des ferme-

tures aux langages LCC offrent une alternative logiquement bien fondée aux méta-appels. Nous montrerons ensuite comment une restriction syntaxique permet d'obtenir un système de module simple et puissant.

Chapitre 3

Langages Concurrents avec Contraintes fondés sur la Logique Linéaire

Sommaire

3.1 Introduction

La classe des langages concurrents avec contraintes (CC) a été introduite par Saraswat dans [SRP91] comme une unification des langages logiques avec contraintes et de la programmation logique concurrente. Dans ce paradigme, les buts PLC sont vus comme des agents concurrents communicant ensembles par l'intermédiaire d'un ensemble commun de contraintes, appelé *store*. Chaque agent est capable de poster une contrainte dans le store – cas des agents *tell* – ou de se synchroniser en attendant qu'une contrainte, appelée dans alors garde, soit impliquée par le store – cas des agents *ask*. Les agents CC on une sémantique logique simple dans la Logique Linéaire (LL) de Girard [FRS01] et cela a conduit à s'intéresser à une extension des langages CC classiques : la classe LCC des langages concurrents avec contraintes basés sur la logique linéaire qui s'affranchit de l'évolution monotone du store imposée par la logique classique, en considérant les contraintes comme des formules de la logique Linéaire (LL) de Girard [Gir87].

Ce chapitre présente essentiellement une version généralisée du cadre théorique des langages LCC proposé par Fages et al. [FRS01]. Comme nous le verrons, cette généralisation consistant à internaliser les déclaration héritées de CC et PLC comme des agents *ask* persistants ne pose pas de gros problèmes théoriques, mais nécessite néanmoins une adaptation de toutes les définitions (syntaxe, sémantique opérationnelle, sémantique logique, monotonie des transition) et des preuves correspondantes.

Dans la première section de ce chapitre, les systèmes de contraintes seront définis de façon habituelle et étudiés. Puis la syntaxe et la sémantique des langages LCC généralisé seront formellement introduites dans les sections 3.3 et 3.4. Nous présenterons, ensuite, dans la section 3.5 quelques propriétés intéressantes de la sémantique opérationnelle. Dans la section 3.6 nous montrerons que les résultats sur la sémantique logique de [FRS01], peuvent être conservés dans le cadre plus général qui est considéré ici. Finalement la dernière section s'intéressera à l'encodage en LCC des traits extra-logiques de PLC présentés à la fin du chapitre précédent.

3.2 Système de Contraintes

Dans la suite, $w, x, y, z, \ldots$ dénoteront des éléments d'un ensemble dénombrable $\mathcal{V}$ de variables et $\vec{x}$ une séquence finie de variables de $\mathcal{V}$. L'ensemble des variables libres d'une formule F sera noté $\mathcal{V}(F)$. De même l'opérateur d'union des multiensembles $\uplus$, appliqué à des ensembles, désigne l'union disjointe, c.-à-d. pour toutes parties E_1 et E_2 d'un même ensemble, $E_1 \uplus E_2$ désigne l'ensemble $E_1 \uplus E_2$ en contraignant E_1 et E_2 a être disjointes.

3.2.1 Définitions

Définition 3.1 (Langage de Contraintes)**.** Une *contrainte linéaire atomique* est une formule construite à partir d'un ensemble $\mathcal{V}$ de variables, d'un ensemble Σ_F de symboles de fonction et d' un ensemble $\Sigma_{\mathcal{C}}$ de symboles de relation.Le *langage de contraintes linéaires* est le plus petit ensemble contenant les contraintes atomiques, l'élément neutre de la conjonction multiplication, $\mathbf{1}$, et clos par la conjonction linéaire multiplicative ($\otimes$), l'exponentiel ($!$) et la quantification existentielle ($\exists$).

Définition 3.2 (Système de contraintes)**.** Un *système de contraintes linéaire* est une paire $(\mathcal{C}, \Vdash_{\mathcal{C}})$ où :

- $\mathcal{C}$ est un langage de contraintes linéaires.
- $\Vdash_{\mathcal{C}}$ est un sous-ensemble de $\mathcal{C} \times \mathcal{C}$ définissant les axiomes non-logiques du système de contraintes.

Dans la suite, nous noterons $\vdash_{\mathcal{C}}$ le plus petit sous-ensemble de $\mathcal{C}^* \times \mathcal{C}$ contenant $\Vdash_{\mathcal{C}}$ et clos par les règles du fragment multiplicatif de la logique linéaire intuitionniste, noté IMLL pour le $\mathbf{1}$, $\mathbf{0}$, $\otimes$, $!$ et $\exists$ (voir l'annexe B pour le calcul des séquents complet) :

$$c \vdash c \qquad \frac{\Gamma, c \vdash d \quad \Delta \vdash c}{\Gamma, \Delta \vdash d} \qquad \vdash \mathbf{1} \qquad \Gamma \vdash \top \qquad \frac{\Gamma \vdash c}{\Gamma, \mathbf{1} \vdash c} \qquad \Gamma, \mathbf{0} \vdash c$$

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

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

3.2.2 Les systèmes de contraintes classiques

Les systèmes de contraintes classiques utilisés dans le chapitre précédent sont des cas particuliers que l'on peut retrouver en utilisant la traduction

standard de la logique classique dans la logique linéaire qui consiste à mettre toute formule sous un bang (!) [Gir87]. Néanmoins, un autre encodage n'utilisant le bang que dans les axiomes non-logiques sera préféré.

Pour encoder formellement un système de contraintes classiques dans un système de contraintes linéaires, nous supposerons qu'un système de contraintes classiques est présenté de façon analogue à un système de contraintes linéaires, c.-à-d. sous forme d'une paire langage de contraintes et relation de déduction.

Nous rappelons ici les règles d'inférence de IL pour les constantes $\top$, $\bot$, le quantificateur $\exists$ et la conjonction $\wedge$:

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

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

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

Définition 3.3. La traduction, notée $\star$, des contraintes classiques vers les contraintes linéaires et la traduction, notée $\diamond$, des contraintes linéaires vers les contraintes classiques sont définies récursivement de la façon suivante :

$$(c \wedge d)^\star = c^\star \otimes d^\star \qquad (\exists x.c)^\star = \exists x.c^\star \qquad \top^* = \mathbf{1} \qquad \bot^* = \mathbf{0} \qquad c^* = c$$

$$(c \otimes d)^\diamond = c^\diamond \wedge d^\diamond \quad (\exists x.c)^\diamond = \exists x.c^\diamond \quad \mathbf{1}^\diamond = \top \quad \mathbf{0}^\diamond = \bot \quad c^\diamond = c \quad (!c)^\diamond = c^\diamond$$

où c est supposée être une contrainte atomique.

De même tout système de contraintes classiques $(\mathcal{C}, \vdash_\mathcal{C})$ est traduit en un système de contraintes linéaires $(\mathcal{C}^\star, \vdash_{\mathcal{C}^\star})$ de la façon suivante :

- $\mathcal{C}^\star$ est le langage de contraintes linéaires obtenu à partir des contraintes atomiques de $\mathcal{C}$.
- $c \Vdash_{\mathcal{C}^\star} !c$ pour toute contrainte atomique de $\mathcal{C}$.
- si $c \Vdash_\mathcal{C} d$ alors $c^\star \vdash_{\mathcal{C}^\star} d^\star$.

Lemme 3.4. *Soient Γ un multiensemble de contraintes linéaires et c une contrainte linéaire.*

$$(i) \quad c \dashv\Vdash_{\mathcal{C}^\star} !c \qquad\qquad (ii) \quad \Gamma, \Delta \vdash_{\mathcal{C}^\star} c \ \ ssi \ \ !\Gamma, \Delta \vdash_{\mathcal{C}^\star} c$$

Preuve. Pour (i), on remarque que la direction $!c \vdash_{\mathcal{C}^\star} c$ est immédiate en utilisant la déréliction. Pour l'autre direction, on procède par induction sur la taille de c :

– $c = d_1 \otimes d_2$:

$$\dfrac{\dfrac{\dfrac{\dfrac{!d_1 \vdash_{\mathcal{C}^\star} d_1 \quad !d_2 \vdash_{\mathcal{C}^\star} d_2}{!d_1, !d_2 \vdash_{\mathcal{C}^\star} (d_1 \otimes d_2)}\, !p}{!d_1, !d_2 \vdash_{\mathcal{C}^\star} !(d_1 \otimes d_2)}\, !p}{d_1, d_2 \vdash_{\mathcal{C}^\star} !(d_1 \otimes d_2)}\, HI}{d_1 \otimes d_2 \vdash_{\mathcal{C}^\star} !(d_1 \otimes d_2)}\, \otimes l$$

– $c = \exists x.d$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{d \vdash_{\mathcal{C}^\star} d}{!d \vdash_{\mathcal{C}^\star} d}\, !d}{!d \vdash_{\mathcal{C}^\star} \exists x.d}\, \exists r}{!d \vdash_{\mathcal{C}^\star} !\exists x.d}\, !p}{d \vdash_{\mathcal{C}^\star} !\exists x.d}\, HI}{\exists x.d \vdash_{\mathcal{C}^\star} !\exists x.d}\, \exists l$$

– $c = !d$: immédiat en utilisant la promotion.

Pour (ii), un sens est toujours direct, grâce à la déréliction. Pour l'autre direction il suffit d'utiliser (i). □

Théorème 3.5 (Correction). *Soient $(\mathcal{C}, \vdash_{\mathcal{C}})$ un système de contraintes classique, Γ un multiensemble de contraintes classiques et c une contrainte classique.*

$$Si \quad \Gamma \vdash_{\mathcal{C}} d \quad alors \quad \Gamma^\star \vdash_{\mathcal{C}^\star} c^\star$$

Preuve. Le résultat est obtenu par une induction sur la preuve π de $\Gamma \vdash_{\mathcal{C}} d$. L'étape d'induction est une conséquence immédiate du lemme précédent. □

Théorème 3.6 (Complétude). *Soient $(\mathcal{C}, \vdash_{\mathcal{C}})$ un système de contraintes classique, Γ un multiensemble de contraintes linéaires et c une contrainte linéaire.*

$$Si \quad \Gamma \vdash_{\mathcal{C}^\star} d \quad alors \quad \Gamma^\diamond \vdash_{\mathcal{C}^\star} c^\diamond$$

Preuve. Par induction sur la preuve de $\Gamma \vdash_{\mathcal{C}^\star} d$. □

3.2.3 Propriétés des systèmes de contraintes linéaires

Quelques propriétés raisonnables sur les systèmes de contraintes linéaires sont définies dans cette section.

Cohérence de l'égalité

Définition 3.7 (Cohérence de l'égalité). Un système de contraintes $(\mathcal{C}, \Vdash_{\mathcal{C}})$ respecte la *cohérence de l'égalité*, si pour tout multiensemble Γ de contraintes *cohérentes* (c.-à-d. $\Gamma \not\vdash_{\mathcal{C}} \mathbf{0}$) et toutes variables distinctes x et y telles que $\Gamma \vdash_{\mathcal{C}} x = y \otimes \top$ on a $\{x, y\} \subset \mathcal{V}(\Gamma)$.

Cette première propriété permet de garantir qu'il n'est pas possible de rendre toutes les variables égales autrement qu'en rendant le store incohérent. Ainsi lorsque des variables fraîches seront introduites, elles pourront être supposées, sans perte de généralité, distinctes à toute autre.

Sous-langage dense

Définition 3.8 (Sous-langage dense)**.** Soit $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes. Un sous-langage $\mathcal{D}$ de $\mathcal{C}$ est *dense*, si pour tout multiensemble Γ de contraintes de $\mathcal{C}$ il existe une contrainte $d \in \mathcal{D}$ telle que :

$$\text{Si} \quad \Gamma \vdash_{\mathcal{C}} d \otimes \top \quad \text{alors} \quad \Gamma \vdash_{\mathcal{C}} \mathbf{0}$$

Exemple 3.9. Soit $(\mathcal{C}, \Vdash_{\mathcal{C}})$, un système de contraintes respectant la *cohérence de l'égalité*. Tout sous-langage de $\mathcal{C}$ contenant l'égalité est dense. En effet pour tout multiensemble Γ de contraintes de $\mathcal{C}$, il suffit de prendre $d = (x = y)$ avec $\{x, y\} \cap \mathcal{V}(\Gamma) = \emptyset$, on a alors $\Gamma \not\vdash_{\mathcal{C}} x = y \otimes \top$ ou $\Gamma \vdash_{\mathcal{C}} \mathbf{0}$.

Sous-langage stable

Définition 3.10 (Sous-langage stable)**.** Soient $(\mathcal{C}, \Vdash_{\mathcal{C}})$, un système de contraintes et $\mathcal{D}$ un sous-langage de $\mathcal{C}$. $\mathcal{C}$ est *stable* sur $\mathcal{D}$, si pour tout multiensemble Γ de contraintes de $\mathcal{D}$ et toute contrainte $c \notin \mathcal{D}$ on a :

$$\text{Si } \Gamma \vdash_{\mathcal{C}} c \quad \text{alors} \quad \Gamma \vdash_{\mathcal{C}} \mathbf{0}$$

Cette propriété garantit qu'à partir de contraintes cohérentes de $\mathcal{D}$ il n'est possible de déduire que des contraintes de $\mathcal{D}$.

Contraintes de synchronisation

Définition 3.11 (Contraintes de synchronisation)**.** Soit $(\mathcal{C}, \Vdash_{\mathcal{C}})$ un système de contraintes. Une *contrainte de synchronisation* ou *token* est une contrainte atomique qui n'apparaît dans aucun axiome non-logique hormis ceux du schéma d'axiomes nécessaire à la définition de la propriété de la *substitution* de l'égalité :

$$p(\vec{x}) \otimes \vec{x} = \vec{y} \vdash_{\mathcal{C}} p(\vec{y})$$

Exemple 3.12. Un exemple caractéristique de système de contraintes basé sur la logique linéaire est la combinaison d'un système de contraintes classique, tel que celui sur les termes d'Herbrand traduit en logique linéaire , avec des contraintes de synchronisation comme $value(x, value)$ – qui associe la valeur $value$ à la référence x – pour encoder les traits impératifs.

Exemple 3.13. Soit $(\mathcal{C}, \Vdash_{\mathcal{C}})$, un système de contraintes respectant la *cohérence de l'égalité* obtenu en combinant un système de contraintes classique traduit en logique linéaire avec des contraintes de synchronisation. Le sous-langage de $\mathcal{C}$ formé des contraintes de synchronisation est dense. En effet pour tout multiensemble de contraintes $\Gamma \in \mathcal{C}^*$, en prendre une contrainte de synchronisation l avec $\mathcal{V}(l) \cap \mathcal{V}(\Gamma) \neq \emptyset$, on a alors $\Gamma \not\vdash_{\mathcal{C}} l \otimes \top$ ou $\Gamma \vdash_{\mathcal{C}} \mathbf{0}$.

Lemme 3.14. *Soient $\mathcal{C}$, $\mathcal{D}$ et $\mathcal{L}$ les langages de contraintes formés respectivement à partir de $\Sigma_{\mathcal{D}} \uplus \Sigma_{\mathcal{P}}$, $\Sigma_{\mathcal{D}}$ et $\Sigma_{\mathcal{P}}$. Si $\mathcal{L}$ est un langage de contraintes ne contenant que des contraintes de synchronisation, alors $\mathcal{C}$ est stable sur $\mathcal{D}$.*

Preuve. Ce résultat se démontre par induction sur la preuve de $\Gamma \vdash c$. □

3.3 Syntaxe

Dans cette section, la version de LCC proposée diffère de celle proposée dans [SL92] et [FRS01]. En effet les déclarations usuelles des langages CC sont abandonnées au profit d'une nouvelle forme d'agent *ask* logiquement équivalent à un *ask* bangué.

Définition 3.15 (Agent LCC)**.** La syntaxe des *agents LCC* est donnée par la grammaire suivante :

$$A ::= A \| A \mid \exists x.A \mid c \mid \forall \vec{x}(c \rightarrow A) \mid \forall \vec{x}(c \Rightarrow A)$$

où les variables $\vec{x}$ sont supposées libres dans les gardes des *ask* (c.-à-d. la contrainte c des agents $\forall \vec{x}(c \rightarrow A)$ et $\forall \vec{x}(c \Rightarrow A)$).

Les opérations possibles pour ces agents sont :

- la mise en parallèle de deux agents par l'opérateur $||$;
- la *localisation* d'une variable par l'opérateur $\exists$;
- l'ajout d'une contrainte c au store, par l'opération *tell* ;
- la suspension sur une certaine contrainte c : l'agent $\forall \vec{x}(c \rightarrow A)$ attend que le store contienne assez d'information pour impliquer la contrainte $c[\vec{x}\backslash\vec{t}]$, lie en conséquence les variables $\vec{x}$ dans l'agent A avant d'exécuter ce dernier et de consommer «logiquement» la contrainte $c[\vec{x}\backslash\vec{t}]$;
- la suspension persistante : l'agent $\forall \vec{x}(c \Rightarrow A)$ n'est rien d'autre qu'un *ask*] classique qui reste actif tout au long de l'exécution du programme.

Dans la syntaxe des agents, les opérateurs $\exists$ et $\forall$ sont, bien entendu, des lieurs de variables ; les variables liées par leur intermédiaire ne seront donc pas considérées comme libres dans l'agent résultant. Les agents seront alors supposés égaux modulo l'α-conversion de leurs variables liées. Par souci de simplicité, dans le cas où un *ask*, persistant ou non, ne lierait aucune variable, nous utiliserons une notation sans quantification universelle. c.-à-d. une notation de la forme $c \rightarrow A$ ou $c \Rightarrow A$.

Le *ask* persistant généralise la notion de déclaration. En effet, une déclaration $p(\vec{y}) :- A$ sera vue comme un *ask* persistant $\forall \vec{y}(p(\vec{y}) \Rightarrow A)$ attendant qu'une contrainte de synchronisation $p(\vec{t})$, représentant l'appel, soit postée dans le store. Un programme sera vu simplement comme la composition parallèle des *ask* persistants représentant les déclarations du programme avec l'agent représentant le but.

Cette présentation de LCC ne fournit pas explicitement l'opérateur de choix (habituellement noté $+$). En effet ce dernier peut être facilement encodé à l'aide d'une contrainte de synchronisation et de *ask* persistants comme suit :

$$A + B = \exists x(choice(x) \,\|\, choice(x) \Rightarrow A \,\|\, choice(x) \Rightarrow B)$$

Cet encodage correspond à l'encodage classique du choix non-déterministe en PLC à l'aide de deux clauses ayant la même tête.

Définition 3.16 (Agent Monotone)**.** L'ensemble des *agents monotones* est le plus petit ensemble clos par composition parallèle qui contient les agents *tell* et *localisation*.

Il a été illustré comment les prédicats à la PLC, peuvent être encodés facilement en LCC grâce à la non-monotonie du store et aux *ask* persistants. Dans la suite, une sous-classe remarquable des agents sera plus particulièrement étudiée. Dans cette classe, définie comme une restriction syntaxique, les *ask* persistants consomment nécessairement des contraintes de synchronisation.

Définition 3.17 (Agent pseudo-stable sur $\mathcal{D}$)**.** Soit $\mathcal{D}$ un sous-ensemble de $\mathcal{C}$. Un agent *LCC* est *D-stable sur* $\mathcal{D}$, si aucune garde de ses aks persistants n'appartient à $\mathcal{D}$.

Pour des raisons de généralité, la définition des agents pseudo-stables est fondée sur une partition du langage de contraintes. Dans la suite, $\mathcal{D}$ sera typiquement l'ensemble des contraintes classiques obtenues par traduction d'un système de contraintes classique, le reste de $\mathcal{C}$ étant constitué de contraintes de synchronisation.

3.4 Sémantique Opérationnelle

3.4.1 Système de transition

La sémantique opérationnelle des langages LCC est définie sur des configurations où le store est distingué des agents.

Définition 3.18 (Configuration)**.** Une *configuration* est un triplet $\langle X;\, c;\, \Gamma\rangle$ où c est une contrainte appelée *store*, Γ est un multiensemble d'agents et X est un ensemble de variables appelées variables cachées.

Définition 3.19 (Équivalence Structurelle)**.** L'équivalence $\equiv$ est la plus petite équivalence sur les configurations satisfaisant la règle de composition parallèle suivante :

$$\langle X;\, c;\, A \| B, \Gamma\rangle \equiv \langle X;\, c;\, A, B, \Gamma\rangle$$

Définition 3.20 (Agent Actif)**.** Un agent LCC A est actif dans une configuration κ s'il existe une configuration de la forme $\langle X;\, c;\, A, \Delta\rangle$ telle que $\kappa \equiv \langle X;\, c;\, A, \Delta\rangle$. Une configuration est une *suspension* si tous ses agents actifs sont des *ask* (persitants ou non). Une configuration est une *suspension persitante* si tous ses agents actifs sont des *ask* persistants.

Définition 3.21. On définit la relation de transition $\longrightarrow$ sur les configurations comme la plus petite relation satisfaisant les règles de la table 3.1.

Dans chacune des règles de la table 3.1, l'agent en gras dans la partie gauche est appelé *agent réductible*. De même l'agent en gras dans la partie sera appelé *réduit* de l'agent réductible correspondant. On supposera que la notion de *réduit* est transitive et réflexive, c.-à-d. que le réduit d'un réduit d'un agent réductible A est un réduit pour A et que A est un de ses propres réduits.

3.4.2 Observables

La sémantique opérationnelle précise dépend comme d'habitude d'un choix d'observable.

Définition 3.22 (Observable)**.** Soit $\mathcal{D}$, un sous-ensemble du langage de contraintes $\mathcal{C}$. Soit κ une configuration LCC telle que $\kappa \xrightarrow{*} \langle \bar{x};\, c;\, \Gamma\rangle$.

- Une contrainte d est une *contrainte accessible* pour κ, si $\exists X. c \vdash_{\mathcal{C}} d \otimes \top$.
- Une contrainte d est une $\mathcal{D}$*-contrainte accessible* pour κ, si d est une contrainte accessible pour A appartenant à $\mathcal{D}$.

Équivalence	$\dfrac{\kappa_1 \equiv \kappa_1' \longrightarrow \kappa_2' \equiv \kappa_2}{\kappa_1 \longrightarrow \kappa_2}$
Tell	$\dfrac{}{\langle X;\, c;\, \boldsymbol{d}, \Gamma\rangle \longrightarrow \langle X;\, c \otimes d;\, \Gamma\rangle}$
Localisation	$\dfrac{y \notin \mathcal{V}(c, \Gamma)}{\langle X;\, c;\, \exists \boldsymbol{y.A}, \Gamma\rangle \longrightarrow \langle X \uplus \{y\};\, c;\, \boldsymbol{A}, \Gamma\rangle}$
Ask	$\dfrac{c \vdash \exists Y.(d[\vec{y}\backslash\vec{t}] \otimes e) \quad Y \cap \mathcal{V}(\Gamma, d) = \emptyset}{\langle X;\, c;\, \forall \boldsymbol{\vec{y}(d \rightarrow A)}, \Gamma\rangle \longrightarrow \langle X \uplus Y;\, e;\, \boldsymbol{A}[\vec{y}\backslash\vec{t}], \Gamma\rangle}$
Ask Persistant	$\dfrac{c \vdash \exists Y.(d[\vec{y}\backslash\vec{t}] \otimes e) \quad Y \cap \mathcal{V}(\Gamma, d) = \emptyset}{\begin{array}{c}\langle X;\, c;\, \forall \boldsymbol{\vec{y}(d \Rightarrow A)}, \Gamma\rangle \longrightarrow \\ \langle X \uplus Y;\, e;\, \boldsymbol{A}[\vec{y}\backslash\vec{t}], \forall \vec{y}(d \Rightarrow A), \Gamma\rangle\end{array}}$

TAB. 3.1 – Relation de transition

- Une contrainte d est un *pseudo-succès* pour κ, si $\exists X.c \vdash_{\mathcal{C}} d$ et si Γ est un multiensemble de *ask* persistants.
- Une contrainte d est *succès* pour κ si d est un *pseudo-succès* pour κ tel que $\langle X;\, c;\, \Gamma\rangle \not\longrightarrow$.
- Une contrainte d est $\mathcal{D}$-*succès* pour κ si d est un succès pour κ appartenant à $\mathcal{D}$.

Dans chacun des cas on dira que l'observable en question est accessible par la dérivation $\kappa \xrightarrow{*} \langle \bar{x};\, c;\, \Gamma\rangle$. On étend naturellement ces définitions à tout agent A en considérant la configuration $\kappa = \langle \emptyset;\, \mathbf{1};\, A\rangle$.

Définition 3.23 (Sémantique opérationnelle)**.** Soit $\mathcal{D}$ un sous-ensemble du langage $\mathcal{C}$ et κ une configuration.

- $\mathcal{O}^{\mathrm{ca}}(\kappa)$ est l'ensemble des contraintes accessibles pour κ.
- $\mathcal{O}^{\mathcal{D}}(\kappa)$ est l'ensemble des $\mathcal{D}$-contraintes accessibles pour κ.
- $\mathcal{O}^{\mathrm{ps}}(\kappa)$ est l'ensemble des pseudo-succès pour κ.
- $\mathcal{O}^{\mathrm{succ}}(\kappa)$ est l'ensemble des succès pour κ.
- $\mathcal{O}^{D\mathrm{succ}}(\kappa)$ est l'ensemble des $\mathcal{D}$-succès pour la configuration κ.

Comme précédemment cette définition est étendue naturellement à un agent A quelconque en considérant la configuration $\kappa = \langle \emptyset;\, \mathbf{1};\, A\rangle$.

3.4.3 Exemples

Le dîner des philosophes

Le test classique d'expressivité des langages concurrents est le dîner des philosophes. Ce problème, énoncé par Dijkstra [Dij71], consiste en N philosophes assis autour d'une table ronde, entre chacun d'eux est disposée une fourchette. Les philosophes passent leur temps à manger ou penser. Le seul problème est que le plat, qu'il leur a été servi, est une sorte de spaghetti particulièrement difficile à saisir qui ne peuvent être mangées qu'à l'aide de deux fourchettes. Comme suggéré dans [BdBP97], ce problème à une solution simple et élégante en LCC.

Ici, l'utilisation des *asks* persistant permet d'éviter une définition récursive pour relancer les agents codant la prise et la pose des fourchette. On obtient ainsi un programme encore plus simple que celle proposé dans [FRS01] est présentée ici. Le système de contraintes dans cet exemple est une combinaison de la traduction en logique linéaire de la contrainte égalité standard sur $\mathbb{N}$ et des contraintes de synchronisation $fork/1$ et $eat/1$ sans autre axiome non-logique que ceux du schéma d'axiome pour la substitution par l'égalité : $c(\vec{x}) \otimes (\vec{x}{=}\vec{y}) \Vdash c(\vec{y})$ pour tout symbole de contrainte c.

Exemple 3.24 (Dîner des Philosophes).

$$
\begin{array}{l}
\forall M, N.recphilo(M,N) \Rightarrow (\\
\quad fork(M) \parallel \\
\quad (fork(I) \otimes fork(I+1 \ mod \ N) \Rightarrow eat(I)) \parallel \\
\quad (eat(I) \Rightarrow fork(I) \otimes fork(I+1 \ mod \ N)) \parallel \\
\quad I \neq N \rightarrow recphilo(M+1,N) \\
)
\end{array}
$$

Par exemple l'exécution de l'agent $recphilo(0, N)$ installera N philosophes et N fourchettes. La séquence des stores d'une dérivation du programme résultant contiendra les différentes phases de réflexion et d'ingurgitation de pâtes, selon l'ordonnancement des agents en parallèle.

On peut noter que, contrairement à un encodage en LCC classique, les agents philosophes simulant la saisie et la libération de deux fourchettes voisines n'ont pas besoin d'être relancés à l'aide d'une déclaration récursive, car leur encodage utilisant un *ask* persistant reste actif tout à long de l'exécution.

La réification de contraintes

La *réification* de contraintes consiste à associer à une contrainte c, une variable booléenne représentant la valeur de vérité de la contrainte. Une fois cette association faite, un solveur de contraintes gérant la réification assure

la cohérence entre la contrainte réifiée et la variable booléenne. Ainsi, dès que ce dernier détecte qu'une contrainte réifiée est vraie – c.-à-d. impliquée par le store – ou fausse – c.-à-d. que sa négation est impliquée par le store – il instancie la variable booléenne en conséquence. Inversement si la variable booléenne est instanciée, le solveur impose la contrainte réifiée ou sa négation, suivant la valeur de la variable.

Le mélange entre contraintes sur les entiers, contraintes booléennes (la valeur *faux* étant représentée par l'entier 0 et la valeur *vrai* par l'entier 1) et la réification offre une très grande puissance d'expression et permet de résoudre certains problèmes de façon élégante et efficace :

Exemple 3.25 (Série Magique). Le problème des séries magiques [Van89] consiste à trouver une suite d'entiers $\{x_0 \dots x_{n-1}\}$ telle que chaque x_i représente le nombre d'occurrences de l'entier i dans la suite. Pour résoudre ce problème, il suffit de poser, pour chaque x_i, la contrainte $x_i = \sum_{j=0}^{n-1} b_{ij}$, où b_{ij} est la variable booléenne de la contrainte réifiée $i = x_j$.

En GNU Prolog, la syntaxe de la réification de contraintes utilise la méta-contrainte `#<=>`. Par exemple `B #<=> (X#=I)` associe `B`, comme variable booléenne, à la contrainte `X#=I`. En supposant que l'on dispose d'un solveur sur les domaines finis, capable de vérifier la statisfiabilité et l'implication de contraintes, la construction GNU-Prolog `B#<=>C`, où `B` est une variable et `C` une contrainte de domaine fini, peut être aisément émulée par quatre *ask*, comme suit (on supposera que `#\` est l'opérateur de négation booléenne et `#=` la contrainte d'égalité sur les domaines finis) :

`C` → `B#=1`,
`#\(C)` → `B#=0`,
`B#=1` → `C`,
`B#=0` → `#\(C)`

Constraint Handling Rules

Le langage *Constraint Handling Rules* (CHR) est un langage introduit par Frühwirth [Frü98] afin de définir simplement des solveurs de contraintes. Il s'agit d'un langage de règles gardées qui récrivent des multiensembles de prédicats, appelés dans ce contexte *contraintes CHR*, en s'appuyant sur un solveur de contraintes natives. Les programmes CHR distinguent donc deux types de contraintes, les contraintes CHR qui sont de simples propositions récrites par les règles et les contraintes natives qu'un solveur de contraintes vérifient satisfiables ou impliquées.

Une règle CHR[1] est une formule de la forme :

$$H_1, \ldots, H_l \texttt{ <=> } G_1, \ldots, G_m | B_1, \ldots, B_n$$

où les H_i, les contraintes de *tête*, sont des contraintes CHR, les G_i, les contraintes de *garde*, sont de contraintes natives et les B_i, les contraintes de corps, sont des contraintes quelconques. Informellement, une telle règle est déclenchée, quand le store contient des contraintes CHR unifiables avec les contraintes de tête et que les contraintes natives précédemment accumulées impliquent la conjonction des contraintes de garde. Dans ce cas les contraintes de têtes sont remplacées par les contraintes de corps.

Il est intéressant de constater que les règles CHR s'interprètent naturellement en LCC. En effet les règles CHR peuvent être vue comme un cas particuliers d'agents LCC. Pour cela on considère les contraintes CHR comme des contraintes de synchronisation, et les contraintes natives comme des contraintes atomiques traduction de contraintes classiques. Ainsi une règle CHR sera vue comme un *ask* persistant de la forme :

$$\forall \vec{x}((H_1 \otimes \cdots \otimes H_i \otimes G_1 \otimes \cdots \otimes G_i) \Rightarrow \exists Y.(G_1 \otimes \cdots \otimes G_j, B_1 \parallel \ldots \parallel B_k))$$

où $\vec{x} = \mathcal{V}(H_1 \ldots H_l, G_1 \ldots G_m)$, $Y = \mathcal{V}(G_1 \ldots G_j, B_1, \ldots, B_m) \setminus \mathcal{V}(H_1 \ldots H_l)$. Cet encodage et les résultats de sémantique logique qui suivent (cf. section 3.6) doivent permettre de retrouver les résultats de [BF05].

Exemple 3.26. En supposant que la contrainte *built-in* égalité `=` est fournie nativement, les règles CHR suivantes définissent une contrainte d'ordre `leq` :

```
leq(X,Y) <=> X=Y | true.
leq(X,Y), leq(Y,X) <=> X=Y.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Ces règles peuvent être représentées par le programme LCC suivant :

$$\forall xy((leq(x,y) \otimes x = y) \Rightarrow \mathbf{1}) \parallel$$
$$\forall xy((leq(x,y) \otimes leq(y,x)) \Rightarrow x = y) \parallel$$
$$\forall xyz((leq(x,y) \otimes leq(y,z)) \Rightarrow leq(x,y) \parallel leq(y,z) \parallel leq(x,z))$$

3.5 Propriétés de la Réduction

Cette section s'intéresse aux propriétés opérationnelles des langages LCC. Certaines de ces propriétés seront essentielles dans les différentes preuves de complétude présentées dans ce mémoire.

[1] seules les règles de simplification sont considérées ici, les autres types de règles pouvant être vus, en première approximation, comme des cas particuliers de simplification.

3.5.1 Monotonie des transitions

L'utilisation de la logique linéaire en lieu et place de la logique classique, fait perdre aux langages LCC la monotonie des contraintes de PLC (proposition 2.7). On conserve tout de même la propriété de *monotonie des transitions*, définie pour LCC avec déclarations dans [FRS01].

Proposition 3.27 (Monotonie des transitions)**.**
Si $\langle X; c; \Gamma\rangle \longrightarrow \langle X \uplus Y; d; \Delta\rangle$ *alors :*

1. *Pour tout ensemble de variables* Z *tel que* $Z \cap Y = \emptyset$ *on a* $\langle X \uplus Z; c; \Gamma\rangle \xrightarrow{*} \langle X \uplus Y \uplus Z; d; \Delta\rangle$.
2. *Pour toutes contraintes* c'*,* e *et tout ensemble de variables* Z *tels que* $c' \vdash_{\mathcal{C}} \exists Z.(c \otimes e)$ *on a* $\langle X; c'; \Gamma\rangle \longrightarrow \langle X \uplus Y \uplus Z; d \otimes e; \Delta\rangle$.
3. *Pour tout multiensemble d'agents* Σ *on a* $\langle X; c; \Gamma, \Sigma\rangle \xrightarrow{*} \langle X \uplus Y; e; \Delta, \Sigma\rangle$.
4. *Pour toute substitution* σ *dont le domaine est disjoint de l'ensemble* Y*, on a* $\langle X; c\sigma; \Gamma\sigma\rangle \xrightarrow{*} \langle X \uplus Y; e\sigma; \Delta\sigma\rangle$.

Preuve. Les différents résultats se démontrent par induction sur la longueur de la dérivation.

- Pour la règle d'*équivalence*, il suffit de remarqué que si $\langle X; c; \Gamma\rangle \equiv \langle X'; c'; \Gamma'\rangle$ alors $\langle X; c\sigma; \Gamma\sigma\rangle \equiv \langle X'; c'\sigma; \Gamma'\sigma\rangle$.
- On constate que la règle *tell*, peut être reproduite dans tous les cas de figure, car elle n'a aucune condition d'application.
- Pour la règle de *localisation*, on se contente d'utiliser l'α-conversion pour que la variable introduite ne soit pas dans $\mathcal{V}(\Sigma, A, \Delta\sigma, c\sigma, X) \cup Z$.
- Pour les règles *ask*, on s'intéresse aux différents cas un à un : Pour le cas *1* il suffit de supposer, sans perte de généralité, que les variables introduites par le *ask* ne contienne pas z. Pour le cas *2* on constate simplement que si $c \vdash_{\mathcal{C}} \exists Y.(d \otimes e)$ et $c' \vdash \exists Z.(c \otimes c'')$ alors $c' \vdash_{\mathcal{C}} \exists ZY.(d \otimes e \otimes c'')$. Le cas *3* est immédiat car les conditions d'application des règles *ask* ne s'intéressent pas aux autre agents présents dans la configuration. Pour le cas *4* on constate que si le domaine de σ est disjoint de Y et si $c \vdash_{\mathcal{C}} \exists Y.(d \otimes e)$ alors $c\sigma \vdash_{\mathcal{C}} \exists Y.(d\sigma \otimes e\sigma)$. □

3.5.2 Stratégie de sélection des agents

La sémantique des langages LCC telle que définie précédemment est hautement indéterministe. Cependant, les résultats de cette section montrent qu'il est possible de restreindre l'analyse à un sous-ensemble de toutes les dérivations possibles en conservant une complétude vis-à-vis des observables précédemment introduits.

Définition 3.28 (Stratégie de Sélection des Agents)**.** Une *stratégie de sélection des Agents* est une fonction qui pour toute dérivation retourne un agent réductible d'une configuration équivalente à la dernière configuration de cette dérivation.

Une dérivation est dite *engendrée* par une stratégie de sélection S, si tout les choix de sélection des agents à réduire dans la dérivation ont été fait suivant S, c.-à-d. que si la dérivation est de la forme :

$$\kappa_0 \longrightarrow \kappa_1 \longrightarrow \ldots \longrightarrow \kappa_n \longrightarrow \ldots$$

alors pour chaque $i \geq 0$ l'agent sélectionné dans l'état κ_i est :

$$S(\kappa_0 \longrightarrow \kappa_1 \longrightarrow \ldots \longrightarrow \kappa_i).$$

Une configuration κ' est accessible à partir d'une configuration κ via une stratégie S s'il existe une dérivation entre κ et κ' générée par S. Par extension un observable c pour κ est accessible via une stratégie S, si c est un observable pour une configuration accessible à partir κ.

A la différence de PLC, l'évaluation de LCC est, bien entendu, dépendante de la stratégie de sélection des agents. On retrouve cependant partiellement le résultat de la proposition 2.9.

Lemme 3.29 (Lemme de permutation)**.** *Soit $\kappa \equiv \langle X;\, c;\, A, B, \Gamma\rangle$ une configuration telle que A soit un agent réductible dans κ et B un agent actif et monotone dans κ. Si $\kappa \longrightarrow \kappa_1 \longrightarrow \kappa'$ est une transition construite en sélectionnant A puis B, alors il existe une dérivation $\kappa \longrightarrow \kappa_2 \longrightarrow \kappa'$ construite en sélectionnant d'abord B puis A.*

Preuve. On constate tout d'abord que $\langle X;\, c;\, A, d, \Gamma\rangle \longrightarrow \langle X;\, c \otimes d;\, A, \Gamma\rangle$ et $\langle X;\, c;\, A, \exists x.B', \Gamma\rangle \longrightarrow \langle X \uplus \{x\};\, c;\, A, B, \Gamma\rangle$ (avec $x \notin \mathcal{V}(c, a, \Gamma)$) sont deux réductions possibles. On conclut en appliquant la monotonie des transitions de LCC. □

Définition 3.30 (Stratégie monotone d'abord)**.** Une stratégie qui sélectionne en priorité les agents monotones par rapport aux *ask* (persistants ou non) est dite *monotone d'abord*. Une famille $\mathbb{S}$ de stratégies monotone d'abord est *homogène* si elle respecte les deux conditions suivantes :

- pour toute stratégie S_1 et S_2 de $\mathbb{S}$, toute dérivation $\kappa_0 \longrightarrow \ldots \longrightarrow \kappa_n$ telle que κ_n contienne des agents monotones, $S_1(\kappa_0 \longrightarrow \ldots \longrightarrow \kappa_n) = S_2(\kappa_0 \longrightarrow \ldots \longrightarrow \kappa_n)$.
- pour toute dérivation $\kappa_0 \longrightarrow \ldots \longrightarrow \kappa_n$ telle que κ_n est une suspension et tout *ask* A (persistant ou non) actif dans κ_n, il existe une stratégie $S \in \mathbb{S}$ telle que $S(\kappa_0 \longrightarrow \ldots \longrightarrow \kappa_n) = A$.

Proposition 3.31. (Indépendance partielle vis-à-vis de la stratégie de Sélection) *Soient R une stratégie de sélection des agents et $\mathbb{S}$ une famille de stratégies monotone d'abord homogène. Toute suspension et toute contrainte accessible pour une configuration κ via R est accessible via une stratégie de $\mathbb{S}$.*

Preuve. Le cas des suspensions est une conséquence immédiate du lemme de permutation. Pour le cas des contraintes accessibles posons que $\kappa \longrightarrow \langle X; c; \Gamma\rangle$. On remarque qu'il est possible de dériver $\langle X; c; \Gamma\rangle$ en utilisant uniquement les règles *tell* et *localisation* vers une suspension de la forme $\langle Y; d; \Delta\rangle$. D'après le cas précédent, on sait que cette suspension est accessible via une stratégie de $\mathbb{S}$. On conclut en constatant $\exists Y.d \vdash_{\mathcal{C}} \exists X.c \otimes \top$. □

3.5.3 Dérivation plus générale

Dans cette sous-section nous montrons qu'il est possible, sans perte de généralité, d'observer uniquement les dérivations dans lesquelles les réduction de *asks* (persistant ou non) affaiblissent le moins possible le store.

Définition 3.32 (Dérivation Plus Générale)**.** Soient κ et κ' deux configurations. κ *est plus générale* que κ' si il existe $\langle X; c; \Gamma\rangle \equiv \kappa$ et $\langle X \uplus Y; d; \Gamma\rangle \equiv \kappa'$ telles que $Y \cap \mathcal{V}(\Gamma) = \emptyset$ et $\exists X.c \vdash_{\mathcal{C}} \exists X \exists Y.d$.

Soient $\delta = (\kappa_1 \longrightarrow \kappa_1 \longrightarrow \ldots \longrightarrow \kappa_n)$ et $\delta' = (\kappa'_1 \longrightarrow \kappa'_1 \longrightarrow \ldots \longrightarrow \kappa'_n)$ deux dérivations. δ est *plus générale* que δ' si pour tout $0 \leq i \leq n$, κ_i est plus générale que κ'_1. De façon symétrique on dira, dans les deux cas, que κ' (resp. δ') est *moins générale* que κ (resp. δ).

Proposition 3.33 (Complétude des dérivations plus générales)**.**

1. *Une contrainte ou un pseudo-succès accessible par une dérivation $\kappa \xrightarrow{*} \kappa'$ est accessible par toute dérivation plus générale que $\kappa \xrightarrow{*} \kappa'$.*
2. *Soit $\mathcal{D}$ un sous-langage de $\mathcal{C}$ et κ_1 une configuration pseudo-stable sur $\mathcal{D}$. Si C est stable sur $\mathcal{D}$ alors un $\mathcal{D}$-succès accessible par une dérivation $\kappa_1 \xrightarrow{*} \kappa'_1$ est accessible par toute dérivation plus générale $\kappa_2 \xrightarrow{*} \kappa'_2$ telle que $\mathbf{0}$ ne soit pas accessible pour κ_2.*

Preuve. Le cas des contraintes accessibles et des pseudo-succès est simple ; il suffit de constater que pour tout configuration $\langle X'; c'; \Gamma'\rangle$ plus générale que $\langle X; c; \Gamma\rangle$ si $\exists X.c \vdash_{\mathcal{C}} e \otimes \top$ et $\exists X.c \vdash_{\mathcal{C}} e'$ alors $\exists X'.c' \vdash_{\mathcal{C}} e \otimes \top$ et $\exists X'.c' \vdash_{\mathcal{C}} e'$.

Pour le cas des $\mathcal{D}$-succès, par définition de la stabilité, si une configuration $\langle X'; c'; \Gamma'\rangle$ est plus générale qu'une configuration $\langle X; c; \Gamma\rangle$ alors $(\exists X.c) \in \mathcal{D}$ implique $(\exists X'.c') \in \mathcal{D}$. On conclut en notant que toute configuration plus générale qu'une configuration pseudo-stable est évidemment pseudo-stable. □

3.6 Sémantique Logique

Un des points forts des langages LCC est qu'ils possèdent une sémantique simple en logique linéaire [FRS01, RF97, SL92], ce qui permet entre autre d'utiliser, sur les programmes LCC, des techniques de preuves héritées de la logique linéaire (comme le calcul des séquents ou la sémantique des phases [FRS01]). Dans cette section nous généralisons cette sémantique dans le fragment Multiplicatif de la Logique Linéaire Intuitionniste (IMLL).

Définition 3.34. On peut traduire les agents LCC en formules logiques IMLL de la manière suivante :

$$c^\dagger = c \qquad (\exists x.c)^\dagger = \exists x.c^\dagger \qquad (A \parallel B)^\dagger = A^\dagger \otimes B^\dagger$$

$$(\forall \bar{x}(c \rightarrow A))^\dagger = \forall \bar{x}(c \multimap A^\dagger) \qquad (\forall \bar{x}(c \Rightarrow A))^\dagger = !\forall \bar{x}(c^\dagger \multimap A^\dagger)$$

Cette traduction s'étend naturellement aux multiensembles d'agents : si Γ est le multiensemble $\{A_1, \ldots, A_n\}$ alors $\Gamma^\dagger = A_1^\dagger \otimes \cdots \otimes A_n^\dagger$; si Γ est vide alors $\Gamma^\dagger = \mathbf{1}$. La traduction d'une configuration $(X; c; \Gamma)$ est la formule $\exists X.(c \otimes \Gamma^\dagger)$.

On notera $\vdash_{IMLL(\mathcal{C})}$ la relation de déduction obtenue en ajoutant $\Vdash_{\mathcal{C}}$ à IMLL (voir l'annexe B pour le calcul des séquents complet).

Avant de faire les preuves formelles de la sémantique logique, nous commençons par deux remarques qui nous servirons par la suite dans plusieurs preuves dans ce chapitre et les suivants.

Proposition 3.35. *Un séquent $\Gamma \vdash_{IMLL(\mathcal{C})} c$, tel que Γ est un multiensemble de traductions d'agents (resp. de contraintes) et c une contrainte, est prouvable si et seulement si $\Gamma \vdash_{IMLL(\mathcal{C})} c$ admet une preuve dont tous les sous-arbres ont pour conclusion un séquent de la forme $\Delta \vdash_{IMLL(\mathcal{C})} d$ où Δ est un multiensemble de traductions d'agents (resp. de contraintes) et d une contrainte.*

Preuve. Soit $\Gamma \vdash_{\mathcal{C}} c$ un séquent prouvable. On peut supposer, sans perte de généralité, que les coupures n'apparaissent qu'au niveau des axiomes non-logiques et qu'elles sont donc de l'une des deux formes suivantes :

$$\frac{\vdots}{\Gamma \vdash_{IMLL(\mathcal{C})} d \quad \overline{d \Vdash_{\mathcal{C}} c}}{\Gamma \vdash_{IMLL(\mathcal{C})} c} \qquad \frac{\overline{d \Vdash_{\mathcal{C}} e} \quad \vdots}{\Gamma, d \vdash_{IMLL(\mathcal{C})} c}{\Gamma, e \vdash_{IMLL(\mathcal{C})} c}$$

L'introduction (de bas en haut) d'une de ces règles n'introduit que des séquents dont la formule à droite reste une contrainte. Il en est de même pour l'introduction à gauche de $\multimap$. Les formules à gauche restent quant à elles des sous-formules de traductions d'agents et restent donc de traductions d'agents. □

Proposition 3.36. *Soit $c_0, \ldots, c_n \in \mathcal{C}^*$ un multiensemble de contraintes.*

$$c_1, \ldots, c_n \vdash_{IMLL(\mathcal{C})} c_0 \quad \textit{si et seulement si} \quad c_1, \ldots, c_n \vdash_{\mathcal{C}} c_0.$$

Preuve. La preuve correspondante à la direction "si" est immédiate car $\vdash_{\mathcal{C}}$ est incluse dans $\vdash_{IMLL(\mathcal{C})}$. La direction "seulement si" est un corollaire de la proposition précédente. □

Cette proposition montre que $\vdash_{\mathcal{C}}$ est correcte et complète vis-à-vis de $\vdash_{IMLL(\mathcal{C})}$ sur les contraintes. On se permettra donc, dans la suite, de confondre, par un léger abus de notation, ces deux relations.

3.6.1 Correction

Théorème 3.37 (Correction)**.** *Soient $(X; c; \Gamma)$ et $(Y; d; \Delta)$ deux configurations LCC.*
Si $(X; c; \Gamma) \equiv (Y; d; \Delta)$ alors $(X; c; \Gamma)^\dagger \dashv\vdash_{\mathcal{C}} (Y; d; \Delta)^\dagger$
Si $(X; c; \Gamma) \xrightarrow{*} (Y; d; \Delta)$ alors $(X; c; \Gamma)^\dagger \vdash_{\mathcal{C}} (Y; d; \Delta)^\dagger$

Preuve. On procède par induction sur $\equiv$ et $\xrightarrow{*}$:

- Pour les règles d'*α-conversion*, de *composition parallèle*, de *tell* et d'*équivalence* le résultat est immédiat.
- Pour les règles de *masquage* on note simplement que $\exists x(A \otimes B) \dashv\vdash A \otimes \exists x B$ et que $\exists x A \dashv\vdash A$ si $x \notin \mathcal{V}(A)$.
- Pour les règles *ask* il suffit de remarquer que
  $c \otimes \forall \vec{x}(d \multimap A) \vdash_{\mathcal{C}} e \otimes d[\vec{x}\backslash\vec{t}] \otimes \forall \vec{x}(d \multimap A) \vdash_{\mathcal{C}} e \otimes A[\vec{x}\backslash\vec{t}]$ et que
  $c \otimes \forall \vec{x}(d \multimap A) \vdash_{\mathcal{C}} e \otimes d[\vec{x}\backslash\vec{t}] \otimes !\forall \vec{x}(d \multimap A) \vdash_{\mathcal{C}} e \otimes A[\vec{x}\backslash\vec{t}] \otimes !\forall \vec{x}(d \multimap A)$, □

3.6.2 Complétude

Lemme 3.38 (Complétude)**.** *Pour tout multiensemble d'agents $\{A_1, \ldots, A_n\}$ et toute contrainte c, si $A_1^\dagger, \ldots, A_n^\dagger \vdash_{\mathcal{C}} c$ alors il existe une dérivation de la forme $\langle \emptyset; \mathbf{1}; A_1, \ldots, A_n \rangle \xrightarrow{*} \langle X; d; \Gamma \rangle$ où $\exists X d \vdash_{\mathcal{C}} c$, Γ est un multiensemble de ask persistants et les variables X sont libres dans c.*

Preuve. Dans cette preuve, $!\Gamma$ et $!\Delta$ seront des notations pour des multiensembles de *ask* persistants. On procède par induction sur la preuve π du séquent $A_1^\dagger, \ldots, A_n^\dagger \vdash_{\mathcal{C}} c$ où les A_i sont des agents et c une contrainte. Grâce à la proposition 3.35 on sait que cette induction a un sens :

- Si π est un axiome : $c \vdash_{\mathcal{C}} d$. Immédiat car $\langle \emptyset; \mathbf{1}; c \rangle \longrightarrow \langle \emptyset; c; \emptyset \rangle$.

– π finit par une coupure :

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c \quad \overline{c \vdash_{\mathcal{C}} d}}{\Gamma^\dagger \vdash_{\mathcal{C}} d} \qquad \text{ou} \qquad \frac{\overline{c \vdash_{\mathcal{C}} c'} \quad \Gamma^\dagger, c' \vdash_{\mathcal{C}} d}{\Gamma^\dagger, c \vdash_{\mathcal{C}} d}$$

Le premier cas est immédiat. Pour le deuxième, par hypothèse d'induction, on sait que $\langle \emptyset; \mathbf{1}; \Gamma, c' \rangle \xrightarrow{*} \langle X; d'; !\Delta \rangle$ tel que $\exists X d' \vdash_{\mathcal{C}} d$ et $X \cap \mathcal{V}(d) = \emptyset$. Dans ce cas, on constate que la règle *tell* qui réduit l'agent correspondant c' peut être appliquée à l'agent correspondant à c sans modifier le store. En effet si $c \vdash_{\mathcal{C}} c'$ et $c' \otimes e \vdash_{\mathcal{C}} e'$ alors $c \otimes e \vdash_{\mathcal{C}} e'$.

– π finit par une introduction à gauche de $\mathbf{1}$:

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, \mathbf{1} \vdash_{\mathcal{C}} c}$$

On remarque que $\langle \emptyset; \mathbf{1}; \mathbf{1}, \Gamma \rangle \longrightarrow \langle \emptyset; \mathbf{1}; \Gamma \rangle$. Or, par hypothèse d'induction, on sait que $\langle \emptyset; \mathbf{1}; \Gamma \rangle \xrightarrow{*} \langle \emptyset; d; !\Delta \rangle$ tel que $\exists X d \vdash_{\mathcal{C}} c$ et $X \cap \mathcal{V}(c) = \emptyset$. D'où $\langle \emptyset; \mathbf{1}; \mathbf{1}, \Gamma \rangle \xrightarrow{*} \langle \emptyset; d; !\Delta \rangle$.

– π finit par une introduction de $\mathbf{0}$: trivial

– π finit par une introduction à gauche de $\otimes$:

$$\frac{\Gamma^\dagger, A, B \vdash_{\mathcal{C}} c}{\Gamma^\dagger, A \otimes B \vdash_{\mathcal{C}} c}$$

Il y a deux sous-cas :

  – Soit $A \otimes B$ est la traduction d'une composition parallèle de deux agents, auquel cas il suffit d'utiliser la règle de *composition parallèle*.

  – Soit $A \otimes B$ est la traduction d'une contrainte de la forme $c \otimes d$, auquel cas il suffit de noter que les deux agents $\langle \emptyset; \mathbf{1}; c, d, \Gamma \rangle$ et $\langle \emptyset; \mathbf{1}; c \otimes d, \Gamma \rangle$ ont les même pseudo-succès.

– π finit par une introduction à droite de $\otimes$:

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c \quad \Delta \vdash_{\mathcal{C}} d}{\Gamma^\dagger, \Delta^\dagger \vdash_{\mathcal{C}} c \otimes d}$$

Par hypothèse d'induction, on sait que $\langle \emptyset; \mathbf{1}; \Gamma \rangle \xrightarrow{*} \langle X; c'; !\Gamma' \rangle$ et $\langle \emptyset; \mathbf{1}; \Delta \rangle \xrightarrow{*} \langle Y; d'; !\Delta' \rangle$ avec $\exists X c' \vdash_{\mathcal{C}} c$, $\exists Y d' \vdash_{\mathcal{C}} d$,$X \cap \mathcal{V}(c) = \emptyset$ et $Y \cap \mathcal{V}(d) = \emptyset$. Grâce à la monotonie de $\longrightarrow$ il est possible d'inférer que $\langle \emptyset; \mathbf{1}; \Gamma, \Delta \rangle \xrightarrow{*} \langle X; c'; \Gamma', \Delta \rangle \xrightarrow{*} \langle X; c' \otimes d'; \Gamma', \Delta' \rangle$. Pour conclure il suffit de noter que si $\exists X c' \vdash_{\mathcal{C}} c$ et $\exists X d' \vdash_{\mathcal{C}} d$ alors $\exists X Y (c' \otimes d') \vdash c \otimes d$ si $X \cap \mathcal{V}(c) = \emptyset$ et $Y \cap \mathcal{V}(d) = \emptyset$.

– π finit par une introduction à gauche de $\exists$:

$$\frac{\Gamma^\dagger, A \dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, \exists x.A^\dagger \vdash_{\mathcal{C}} c} x \notin \mathcal{V}(\Gamma, c)$$

Par hypothèse d'induction, on sait que $\langle \emptyset; \mathbf{1}; A, \Gamma \rangle \xrightarrow{*} \langle X; d; !\Delta \rangle$) avec $\exists X.d \vdash_{\mathcal{C}} c$ et $X \cap \mathcal{V}(c) = \emptyset$ et donc par monotonie de $\longrightarrow$, $\langle x; \mathbf{1}; A, \Gamma \rangle \xrightarrow{*} \langle X', x; d[X \backslash X']; !\Delta[X \backslash X'] \rangle$) avec, sans perte de généralité, $X' \cap \mathcal{V}(c) = \emptyset$. Puisque $x \notin \mathcal{V}(\Gamma)$, nous déduisons que $\langle \emptyset; \mathbf{1}; \exists x.A, \Gamma \rangle \equiv \langle x; \mathbf{1}; A, \Gamma \rangle$. On conclut en notant que si $\exists X.d \vdash_{\mathcal{C}} c$ et $X' \cap \mathcal{V}(c) = \emptyset$ alors $\exists X'.d[X \backslash X'] \vdash_{\mathcal{C}} c$.

– π finit par une introduction à droite de $\exists$: immédiat.

– π finit par une règle d'introduction à gauche de $\multimap$:

$$\frac{\Gamma \vdash d \quad \Delta, A \vdash c}{\Gamma, \Delta, d \multimap A \vdash_{\mathcal{C}} c}$$

Par hypothèse d'induction, on sait que $\langle \emptyset; \mathbf{1}; \Gamma \rangle \xrightarrow{*} \langle X; d'; !\Gamma' \rangle$ et $\langle \emptyset; \mathbf{1}; \Delta \rangle \xrightarrow{*} \langle Y; c'; !\Delta' \rangle$. Grâce à l'$\alpha$-équivalence, on suppose, sans perte de généralité, que $X \cap Y = \emptyset$. Il suffit maintenant d'utiliser la monotonie de $\xrightarrow{*}$ et la règle *ask* pour conclure que $\langle \emptyset; \mathbf{1}; \Gamma, \Delta, d \to A \rangle \xrightarrow{*} \langle X; d'; !\Gamma', \Delta, d \to A \rangle \longrightarrow \langle X; \mathbf{1}; !\Gamma', \Delta, A \rangle \xrightarrow{*} \langle X, Y; c'; !\Gamma', !\Delta', A \rangle$.

– finit par une règle d'introduction à gauche de $\forall$:

$$\frac{\Gamma, A[\vec{x} \backslash \vec{t}] \vdash c}{\Gamma, \forall \vec{x} A \vdash c}$$

Cette situation ne se présente que quand A est la traduction d'un *ask* $d \to B$. On vérifie alors aisément que si $d[\vec{x} \backslash \vec{t}] \to B[\vec{x} \backslash \vec{t}]$ permet une dérivation où ce *ask* disparaît (puisqu'il ne peut apparaître dans c), alors $\forall \vec{x}(d \to B)$ permet la même.

– π finit avec une *déréliction* : Grâce à la remarque sur la permutabilité des règle, faite en préliminaire, nous savons qu'il n'y a que deux sous-cas :

$$\frac{\Gamma^\dagger, d^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !d^\dagger \vdash_{\mathcal{C}} c} \qquad \text{ou} \qquad \frac{\Gamma^\dagger, \forall \vec{z}.(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !\forall \vec{z}.(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}$$

Dans le premier cas ,le résultat est évident, il suffit de se rappeler que $!d \vdash d$. Dans le second cas, on sait, par hypothèse d'induction, $\langle \emptyset; \mathbf{1}; \Gamma, \forall \vec{x}(d \to A) \rangle \xrightarrow{*} \langle X; c'; !\Gamma' \rangle$ avec $\exists X.c' \vdash_{\mathcal{C}} c$ et $X \cap \mathcal{V}(c) = \emptyset$. Ainsi en remplaçant dans cette dérivation l'application de la règle *ask* qui réduit l'agent $\forall \vec{x}(d \to A)$ (on sait qu'une telle réduction est nécessaire sinon $!\Gamma'$ ne serait pas un multiensemble de *ask* persistants) par la

règle *ask persistant* nous obtenons la dérivation $\langle \emptyset;\ \mathbf{1};\ \Gamma, \forall \vec{x}(d \Rightarrow A)\rangle \xrightarrow{*} \langle X;\ c';\ \forall \vec{x}(d \Rightarrow A), !\Gamma'\rangle$.

– π finit avec une *promotion* :

$$\frac{!\Gamma^\dagger \vdash_{\mathcal{C}} !c}{!\Gamma^\dagger \vdash_{\mathcal{C}} c}$$

Par induction, $\langle \emptyset;\ \mathbf{1};\ !\Gamma\rangle \xrightarrow{*} (X; d; !\Gamma')$ avec $\exists X. d^\dagger \vdash_{\mathcal{C}} c$ et $X \cap \mathcal{V}(c) = \emptyset$. Pour conclure on se rappelle que $!c \vdash_{\mathcal{C}} c$.

– π finit par un *affaiblissement* : Il y a deux sous-cas

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !d^\dagger \vdash_{\mathcal{C}} c} \qquad \text{ou} \qquad \frac{\Gamma^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !\forall \bar{x}(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}$$

Dans le premier on note que $(\emptyset; \mathbf{1}; !\mathbf{d}, \mathbf{\Gamma}) \xrightarrow{*} (\emptyset; !\mathbf{d}; \mathbf{\Gamma})$ et $!d \vdash_{\mathcal{C}} \mathbf{1}$. Le second cas est immédiat car la formule $!\forall \bar{x}(d^\dagger \multimap A^\dagger)$ est la traduction d'un *ask* persistant.

– π finit avec une *contraction* : Il y a deux sous-cas

$$\frac{\Gamma^\dagger, !d^\dagger, !d^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !d^\dagger \vdash_{\mathcal{C}} c} \qquad \text{ou} \qquad \frac{\Gamma^\dagger, !\forall \bar{x}(d^\dagger \multimap A^\dagger), !\forall \bar{x}(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !\forall \bar{x}(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}$$

Pour le premier cas il est suffisant de constater que $!d \dashv\vdash !d \otimes !d$. Le second cas est trivial, car toute contrainte consommée par deux *ask* persistant identiques mis en parallèle peut être consommée par un *ask* persistant isolé. □

Proposition 3.39 (Observation des pseudo-succès)**.** *Pour tout agent LCC A on a :*

$$\mathcal{O}^{ps}(A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{\mathcal{C}} c\}$$

Preuve. Une inclusion est immédiate : il suffit d'appliquer le théorème de correction et de noter que $!\Gamma, c \vdash c$. L'autre inclusion est une conséquence directe du lemme de complétude. □

Théorème 3.40 (Observation des contraintes accessibles)**.** *Soit $\mathcal{D}$ un sous-ensemble du langage $\mathcal{C}$. Pour tout agent LCC A on a :*

$$\mathcal{O}^{ca}(A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{\mathcal{C}} c \otimes \top\} \qquad \mathcal{O}^{\mathcal{D}}(A) = \{c \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}} c \otimes \top\}$$

Preuve. Nous montrons ici le résultat pour les contraintes accessibles, le cas des $\mathcal{D}$-contraintes accessibles étant une conséquence triviale de ce résultat. Une inclusion est toujours immédiate : il suffit d'appliquer le théorème de

correction et de noter que $\Gamma, c \vdash c \otimes \top$. Concernant l'autre inclusion (celle correspondante à la complétude), on remarque que si $A^\dagger \vdash c \otimes \top$ alors la preuve commence (de bas en haut) par des éliminations à gauche, correspondant exactement aux réductions de A, puis par une élimination des $\exists$ des deux cotés et finalement une séparation du membre de droite en deux formules, correspondant à deux configurations κ et κ', telles que $\kappa^\dagger \vdash_{\mathcal{C}} c$ et $\kappa'^\dagger \vdash_{\mathcal{C}} \Gamma \otimes \top$. On peut alors utiliser la proposition 3.39 sur κ, ainsi c est une contrainte accessible pour κ. On conclue, à l'aide de la monotonie de $\longrightarrow$. □

Après avoir caractérisé logiquement les pseudo-succès et les contraintes accessibles pour un agent LCC($\mathcal{C}$) quelconque, il serait souhaitable de caractériser les stores finals. Hélas, à cause de l'emploi du bang (!) dans notre traduction des *ask* persistants, cela nous est impossible pour tout agent contenant des *ask* persistants. En effet, la règle d'*affaiblissement* pour le ! permet d'oublier une formules correspondant à la traduction d'un *ask* persistant avant qu'il se soit synchroniser sur toutes les contraintes qu'il aurait été capable de consommer. Néanmoins en supposant quelques propriétés sur les contraintes consommées par les *ask* persistants nous pouvons caractériser un sous-ensemble intéressant des succès.

Théorème 3.41 (Observations des $\mathcal{D}$-succès)**.** *Soit $\mathcal{D}$ un sous-langage de $\mathcal{C}$ et A un agent LCC pseudo-stable sur $\mathcal{D}$ qui n'admet pas* **0** *comme contrainte accessible. Si C est stable sur $\mathcal{D}$ alors :*

$$\mathcal{O}^{\mathcal{D}-succ}(A) = \{d \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}} d\}$$

Preuve. L'inclusion correspondante à la correction est évidente.

D'après la proposition 3.39, nous savons que pour toute contrainte de $\mathcal{D}$, il existe une dérivation $\langle \emptyset; \mathbf{1}; A\rangle \xrightarrow{*} \langle X; \mathbf{d}'; !\Gamma\rangle$ où $!\Gamma$ est un multiensemble de *ask* persistants et $\exists X. d' \vdash_{\mathcal{C}} d$. Pour prouver la deuxième inclusion, correspondante à la complétude, il nous suffit de montrer que $\langle X; \mathbf{d}'; !\Gamma\rangle$ est irréductible. Supposons, par contraposition, que cette configuration est réductible, c'est à dire qu'il existe un *ask* persistant dans $!\Gamma$ de la forme $\forall \vec{x}(c \Rightarrow B)$ tel quel $d \vdash c \otimes e$ pour une certaine contraintes e. On déduit que $d \vdash_{\mathcal{C}} c \otimes \top$, ce qui contredit les hypothèses car $c \notin D$ est $\mathcal{C}$ est borné par $\mathcal{D}$. □

3.7 la PLC et ses Traits Extra-logiques

3.7.1 Les clauses comme des agents

Nous avons déjà montré que les *ask* persistants de LCC généralisent la notion de déclaration usuelle des langages CC et donc des clauses PLC. Dans les exemples suivants, par souci de clarté, nous utiliserons la notation des clauses pour représenter des *ask* persistants. En d'autres termes $p(\vec{t}) :- A$ sera une notation simplifié de $\forall \vec{y}(p(\vec{y}) \Rightarrow \exists \vec{x}.\vec{y}{=}\vec{t} \parallel A)$ où $\vec{x} = f(A)$ et $p(\vec{y})$ est supposé être une contrainte de synchronisation dans $(\mathcal{C}, \Vdash_{\mathcal{C}})$. De même on utilisera, la police "`machine à écrire`" pour écrire les programmes et les identifiants commençant par une lettre majuscule pour représenter les variables.

3.7.2 Les termes mutables

Grâce aux contraintes de synchronisation, il est possible d'encoder très simplement les termes mutables présentés au chapitre précédent.

Exemple 3.42 (Encodage des termes mutables).

`create_mutable(V,M):-mutable(M,V).`
`get_mutable(V,M):-∀O(mutable(M,O) → mutable(M,O),O = V).`
`update_mutable(V,M):-∀O(mutable(M,O) → mutable(M,V)).`

Pour cet encodage on suppose simplement que `mutable/2` est une contrainte de synchronisation. L'initialisation (`create_mutable/2`) consiste à ajouter au store la contrainte `mutable(M,Val)` qui associe à une variable M, représentant le terme mutable, une valeur `V`. La lecture de la valeur courante associée à `M` (`get_mutable/2`), ce fait par l'intermédiaire d'un *ask*; il est alors nécessaire de poster à nouveau la contrainte de synchronisation `mutable(M,V)` qui a été consommée. La mise à jour d'un mutable (`update_mutable/2`), est analogue à la lecture, excepté que la contrainte postée en fin, associe `M` à la nouvelle valeur.

3.7.3 Les variables globales

L'encodage des variables globales est très similaire à celui des termes mutables. La seule différence notable est que la valeur est associée à un atome – c.-à-d. une constante de Σ_F – au lieu d'une variable. Il est ainsi possible d'y accéder globalement en connaissant simplement le nom de cet atome. Dans le programme suivant, les prédicats `g_init`, `g_link` et `g_read`

ont un rôle analogue aux prédicats `create_mutable/2`, `get_mutable/2` et `update_mutable/2` de l'exemple précédent.

Exemple 3.43 (Encodage des variable globales).

$\texttt{g\_init}(\texttt{K},\texttt{V})\texttt{:-global}(\texttt{K},\texttt{V}).$
$\texttt{g\_link}(\texttt{K},\texttt{V})\texttt{:-}\forall\texttt{O}(\texttt{global}(\texttt{K},\texttt{O}) \rightarrow \texttt{global}(\texttt{K},\texttt{O}),\texttt{O} = \texttt{V}).$
$\texttt{g\_read}(\texttt{K},\texttt{V})\texttt{:-}\forall\texttt{O}(\texttt{global}(\texttt{K},\texttt{O}) \rightarrow \texttt{global}(\texttt{K},\texttt{V})).$

3.7.4 Les méta-appels

Formellement la construction `call` est encodée en LCC, en supposant que pour tout symbole de prédicat `p` appartenant Σ_p, il existe un *ask* persistant $\forall\tilde{\texttt{X}}(\texttt{call}(\texttt{p}(\tilde{\texttt{X}})) \Rightarrow \texttt{p}(\tilde{\texttt{X}}))$ actif dans le programme.

Suivant la sémantique de LCC, cette implémentation, n'est pas exactement équivalente à la sémantique effective du `call` Prolog. En effet alors que ce dernier émet une exception quand son paramètre n'est pas instancié, l'implémentation proposée dans cette section suspend simplement l'exécution du `call`.

3.7.5 Les assertions dynamiques de clauses

L'appel au prédicat Prolog $\texttt{assert}((\texttt{p}(\texttt{X}_1,\ldots,\texttt{X}_\texttt{n}) : -\texttt{Body}))$ peut être interprété en LCC comme l'agent $\forall\texttt{X}_1,\ldots,\texttt{X}_\texttt{n}((\texttt{p}(\texttt{X}_1,\ldots,\texttt{X}_\texttt{n}) \Rightarrow \texttt{Body})$. De plus le renommage des variables que `assert` fait de façon transparente peut être simplement émulé par une quantification explicite fournie par l'opérateur LCC $\exists$. Il doit être noté cependant que cette implémentation du `assert` est backtrackante, c.-à-d. que les clauses asserées seront retirées durant le backtrack.

3.7.6 Les variables attribuées

La lecture et la mise à jour d'un attribut d'une variable s'encodent de façon similaire aux termes mutables. Dans le programme suivant les prédicats `init_att`, `put_att` et `get_att` ont un rôle symétrique aux prédicats respectifs `create_mutable/2`, `get_mutable/2` et `update_mutable/2` de l'exemple 3.42, `attribut` étant une contrainte de synchronisation permettant d'attacher à une variable son attribut.

$\texttt{init\_att}(\texttt{X},\texttt{A})\texttt{:-attribut}(\texttt{X},\texttt{V}).$
$\texttt{put\_att}(\texttt{X},\texttt{A})\texttt{:-}\forall\texttt{O}(\texttt{attribut}(\texttt{X},\texttt{O}) \rightarrow \texttt{attribut}(\texttt{X},\texttt{O}),\texttt{A} = \texttt{V}).$
$\texttt{get\_att}(\texttt{X},\texttt{A})\texttt{:-}\forall\texttt{O}(\texttt{global}(\texttt{X},\texttt{O}) \rightarrow \texttt{attribut}(\texttt{X},\texttt{A})).$

Cependant, à la différence des termes mutables, on supposera que l'agent ci-dessous est placé en composition parallèle avec le programme. Cet agent a pour rôle de déclencher le prédicat `verify_attribut` défini par l'utilisateur quand deux variables attribuées sont unifiées (cas du premier *ask* persistant) ou quand une variable attribuée est unifiée avec un terme instancié.

$$\begin{array}{l}\forall \mathtt{X}, \mathtt{V_1}, \mathtt{V_2}((\mathtt{attribut}(\mathtt{X}, \mathtt{V_1}) \otimes \mathtt{attribut}(\mathtt{X}, \mathtt{V_2})) \Rightarrow \\ \qquad (\mathtt{attribut}(\mathtt{X}, \mathtt{V_1}), \mathtt{attribut}(\mathtt{X}, \mathtt{V_2}), \mathtt{verify\_attributs}(\mathtt{X}, \mathtt{X}, \mathtt{G}))) \parallel \\ \forall \mathtt{X}, \mathtt{V}((\mathtt{attribut}(\mathtt{X}, \mathtt{V_1}) \otimes \mathtt{nonvar}(\mathtt{X})) \Rightarrow \\ \qquad (\mathtt{attribut}(\mathtt{X}, \mathtt{V_1}), \mathtt{verify\_attributs}(\mathtt{X}, \mathtt{X}, \mathtt{G})))\end{array}$$

On notera que grâce à cet encodage des coroutines de Prolog en LCC et grâce à l'encodage du `freeze` utilisant les attributs (présenté dans l'exemple 2.19) on retrouve cette coroutine facilement en LCC.

3.8 Discussion

Dans le cadre restreint de la classe des langages CC classiques avec déclarations tel que présenté par Saraswat [Sar93b], le quantificateur universel dans les *ask* n'est pas toujours nécessaire. Par exemple, dans le cas où le domaine des variables est finis, il est possible de remplacer sans perte de généralité un *ask* de la forme $\forall \vec{x}(c \rightarrow A)$ par la construction non universellement quantifiée de la forme $(\exists \vec{x}.c) \rightarrow \exists \vec{x}.(c \parallel A)$. En effet bien que ces constructions ont des comportements opérationnels distincts, elles possèdent le même ensemble de solutions (c.-à-d. les même observables). Le $\forall$ est néanmoins indispensable pour encoder les déclarations comme des agents (cf sous-section 3.7.1), les traits impératifs (cf sous-section 3.7.2 et 3.7.3) et les fermetures (cf chapitre suivant).

Dans le cadre du projet SiLCC, l'utilisation de concurrence permise par les langages CC/LCC est principalement motivée par le gain en expressivité qu'elle par rapport aux langages de programmation logique avec contraintes classiques. En effet, la concurrence est très utile pour définir des *threads* implémentant des solveurs de contraintes capables de vérifier, en parallèle à l'exécution courante, la cohérence de l'ensemble des contraintes déposées au court du calcul. Le mécanisme de variables attribuées présenté à la section 2.3.6, fournit assez de concurrence pour réaliser cette fonction, mais ne permet pas, comme dans des langage concurrents tel que Oz, une vraie préemption : un *thread* peut être interrompu par un nouveau *thread* mais il doit attendre que ce dernier ait terminé son exécution, pour pouvoir reprendre son déroulement.

Chapitre 4

Fermeture dans les langages LCC

Sommaire

4.1 Introduction

Ce chapitre illustre le fait que la classe des langages présentée dans le chapitre précédent possède la puissance expressive des fermetures, c.-à-d. , la possibilité de manipuler, comme un objet de premier ordre, du code avec une environnement. Dans ce chapitre nous présentons des codages compositionnels, à la Milner [Mil90], pour deux stratégie d'évaluation du λ-calcul en LCC. Pour cela nous reprenons et formalisons dans un cadre premier ordre des travaux non publiés et non prouvés de Saraswat et Lincoln [SL92][1].

Laneve et Montanari [LM92] ont démontré qu'un tel encodage était déjà possible dans CC. Cet encodage possède néanmoins quelques défauts. Tout d'abord cet encodage utilise des déclarations et n'est donc pas parfaitement compositionel. De plus il nécessite un mécanise complexe de *stream* qui sera

[1]Le cadre formel de ce papier est une extension de LCC basée sur la logique linéaire d'ordre supérieur.

avantageusement remplacé ici par l'utilisation de contraintes de synchronisation.

La correction de ces codages sera démontrée en s'appuyant sur les résultats de sémantique logique. La complétude sera prouvée par l'intermédiaire d'une correspondance entre les réductions des *asks* persistants du codage et la β-réduction du λ-cacul. Ce chapitre ce termine par une présentation d'un encodage du λ-calcul paresseux et une discussion générale sur l'encodage du λ-calcul dans différents paradigmes concurrents.

Exemple

En LCC, le mécanisme de portée des variables et les *ask* persistants rendent possible l'encodage de *fermetures*, c.-à-d. la possibilité de manipuler comme objet de première classe du code et un environnement. Par exemple, l'agent $\forall x(apply(y,x) \Rightarrow (x_{min} \leq x \parallel x \leq x_{max}))$ attend que la contrainte de synchronisation $apply(y,z)$, soit ajoutée au store. D'un point de vue fonctionnel la variable y représente la fonction $\lambda x.(x_{min} \leq x \parallel x \leq x_{max})$ et l'agent $apply(y,z)$ représente l'application de la valeur z à la fonction représentée par y.

Il est alors possible de définir des itérateurs pour pour les structures de données tels que `forall` et `exists` pour les listes de façon plus élégante que dans l'exemple Prolog (example 2.15) :

Exemple 4.1 (Itérateurs de listes). :
$\forall y(forall([],y) \Rightarrow \mathbf{1}) \parallel$
$\forall x_H x_T y(forall([x_H|x_T],y) \Rightarrow (apply(y,x_H) \parallel forall(x_T,y)) \parallel$

$\forall x_H x_T y(exists([x_H|x_T],y) \Rightarrow apply(y,x_H)) \parallel$
$\forall x_H x_T y(exists([x_H|x_T],y) \Rightarrow exists(x_T,y))$

4.2 Le λ-calcul

Définition 4.2 (λ-terme)**.** La syntaxe des *valeurs*, notées U, V, ou W, et des λ-*termes*, notés L, M ou N du lambda calcul est donnée par la grammaire suivante :

$$V ::= x \mid \lambda x.M \qquad\qquad M ::= V|MM$$

où x est une variable appartenant à un ensemble dénombrable $\mathcal{V}_\lambda$. Il existe donc trois types de λ-termes : les variables (x), les *abstractions* ($\lambda x.M$) et les *applications* (MN). On notera Λ l'ensemble des λ-termes.

Rappelons la notion de position dans un λ-terme.

Définition 4.3 (Position)**.** Une *position* pour les λ-termes est une séquence d'entiers compris entre 1 et 2, ϵ désignant la séquence vide. L'ensemble des positions d'un λ-terme est l'ensemble défini récursivement comme :

- $\mathrm{Pos}(x) = \{\epsilon\}$
- $\mathrm{Pos}(\lambda x.M) = \{\epsilon\} \cup \{1.p \mid p \in \mathrm{Pos}(M)\}$
- $\mathrm{Pos}(M_1 M_2) = \{\epsilon\} \cup \{1.p \mid p \in \mathrm{Pos}(M_1)\} \cup \{2.p \mid p \in \mathrm{Pos}(M_2)\}$

Le sous-terme d'un terme M à la position $p \in \mathrm{Pos}(M)$ est le terme $M\,|_p$ défini récursivement comme :

$$M|_\epsilon = M \qquad M_1M_2|_{1.p} = M_1|_p \qquad M_1M_2|_{2.p} = M_2|_p \qquad \lambda x.M|_{1.p} = M|_p$$

Comme le fait Milner pour encoder le λ-calcul dans le π-calcul, une stratégie particulière de réduction du λ-calcul est considérée ici. Il s'agit de la stratégie « appel par valeur », qui représente l'évaluation des langages de type ML[GMM+78].

Définition 4.4. La *réduction en « appel par valeur »* est la plus petite relation $\rightarrow_{\beta_V}$ qui satisfait les règles suivantes :

$$\beta_V : (\lambda x.M)V \rightarrow_{\beta_V} M[x\backslash N]$$

$$APPL : \frac{M \rightarrow_{\beta_V} M'}{MN \rightarrow_{\beta_V} M'N} \qquad APPR : \frac{N \rightarrow_{\beta_V} N'}{MN \rightarrow_{\beta_V} MN'}$$

4.3 Un Codage Compositionnel

Ici, l'ensemble $\mathcal{V}_\lambda$ des variables du λ-calcul, ainsi qu'un ensemble dénombrable $\mathcal{V}_e$ de variables d'encodage seront supposés disjoints et inclus dans l'ensemble $\mathcal{V}$ des variables *LCC*

Définition 4.5 (Codage du λ-calcul avec « appel par valeur »)**.**
Chaque terme M du λ-calcul est codé par $[\![M]\!]$ une fonction des variables vers les agents $\mathrm{LCC}(\mathcal{C}_v)$ et définie récursivement par :

$$[\![x]\!](y) = (x{=}y) \otimes value(y)$$
$$[\![\lambda x.M]\!](y) = \forall xz(apply(y,x,z) \otimes value(x) \Rightarrow [\![M]\!](z)) \parallel value(y))$$
$$[\![MN]\!](y) = \exists zz'.\,(apply(z,z',y) \parallel [\![M]\!](z) \parallel [\![N]\!](z'))$$

où y, z et z' sont supposées appartenir à $\mathcal{V}_e$. De plus $\mathcal{C}_v$ est supposé être le système de contraintes construit à partir des axiomes non-logiques suivants :

1. $x{=}y \Vdash_{\mathcal{C}_v} !x{=}y$ (= est une contrainte classique)

2. $\Vdash_{\mathcal{C}_v} x{=}x$ (= est réflexive)
3. $x{=}y \Vdash_{\mathcal{C}_v} y{=}x$ (= est symétrique)
4. $x{=}y \otimes y{=}z \Vdash_{\mathcal{C}_v} x{=}z$ (= est transitive)
5. $value(x) \Vdash_{\mathcal{C}_v} !value(x)$ ($value$ est une contrainte classique)
6. $value(x) \otimes x{=}y \Vdash_{\mathcal{C}_v} !value(y)$ (substitution pour $value$)
7. $apply(x,y,z) \otimes (x,y,z) = (x',y',z') \Vdash_{\mathcal{C}_v} apply(x',y',z')$ (substitution pour $apply$)

Exemple 4.6. Le codage du λ-terme $M = (\lambda x.xx)y$ est l'agent :

$$
\begin{array}{l}
\exists z_1 z_2 (apply(z_1, z_2, z_\epsilon) \parr \\
\quad \forall x z_{11}.(apply(z_1, x, z_{11}) \Rightarrow \\
\quad\quad \exists z_{111} z_{112}.(apply(z_{111}, z_{112}, z_{11}) \parr \\
\quad\quad\quad x = z_{111} \otimes value(z_{111}) \parr \qquad \} M|_{111} = x \\
\quad\quad\quad x = z_{112} \otimes value(z_{112}) \qquad \} M|_{112} = x \\
\quad\quad) \\
\quad) \parr \\
\quad y = z_2 \otimes value(z_2) \qquad\qquad \left.\begin{array}{c} M|_{11} \\ = \\ (xx) \end{array}\right\} \begin{array}{c} M|_1 \\ = \\ \lambda x.xx \end{array} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} M|_2 = y \\
)
\end{array}
$$

On peut remarquer que chaque sous-terme de M est associé à une variable d'encodage, ici un sous-terme $M|_p$ est associé à une variable z_p.

4.3.1 Correction

Pour prouver la correction du codage présenté précédemment la sémantique logique de LCC est utilisée. En effet, les démonstrations suivantes se contentent de prouver l'implication logique entre les traductions logiques d'un λ-terme et son réduit.

Lemme 4.7. *Soit V une valeur*

(i) $[\![V]\!](y)^\dagger \vdash_{\mathcal{C}_v} 1$

(ii) $[\![V]\!](y)^\dagger \vdash_{\mathcal{C}_v} [\![V]\!](y)^\dagger \otimes [\![V]\!](y)^\dagger$

Preuve. Pour (i) il suffit d'utiliser l'affaiblissement du !. Pour montrer (ii) cela, on utilise les axiomes 1 et 5 si V est une variable et l'axiome 5 et la contraction du ! si V est une fonction. □

Lemme 4.8 (Correction de la substitution)**.**

$$\exists z([\![M[x\backslash z]]\!](y)^\dagger \otimes [\![V]\!](y)^\dagger) \vdash_{\mathcal{C}_v} [\![M[x\backslash V]]\!](y)^\dagger$$

Preuve. On procède par induction sur M.

– $M = x' \neq x$:

$$\dfrac{\dfrac{[\![x']\!](y)^\dagger \vdash_{\mathcal{C}_v} [\![x']\!](y)^\dagger \quad \dfrac{}{[\![V]\!](z)^\dagger \vdash_{\mathcal{C}_v} \mathbf{1}}\ \text{lem 4.7}}{[\![x']\!](y)^\dagger \otimes [\![V]\!](z)^\dagger \vdash_{\mathcal{C}} [\![x']\!](y)^\dagger}\ \otimes\text{-l, cut}}{\exists z'.([\![x']\!](y)^\dagger \otimes [\![V]\!](z)^\dagger) \vdash_{\mathcal{C}_v} [\![x']\!](y)^\dagger}\ \exists\text{-l}$$

– $M = x$:

$$\dfrac{\dfrac{}{value(y) \vdash_{\mathcal{C}_v} !value(y)}\ \text{x-5} \quad \dfrac{\dfrac{[\![V]\!](y)^\dagger \vdash_{\mathcal{C}_v} [\![V]\!](y)^\dagger}{z = y, [\![V]\!](z)^\dagger \vdash_{\mathcal{C}_v} [\![V]\!](y)^\dagger} =}{z = y, !value(y), [\![V]\!](z)^\dagger \vdash_{\mathcal{C}_v} [\![V]\!](y)^\dagger}\ !\text{-w}}{z = y, value(y), [\![V]\!](z)^\dagger \vdash_{\mathcal{C}_v} [\![V]\!](y)^\dagger}\ \otimes\text{-l, cut}}{[\![z]\!](y)^\dagger \otimes [\![V]\!](z)^\dagger \vdash_{\mathcal{C}} [\![V]\!](y)^\dagger}\ \otimes\text{-l, } [\![]\!]}{\exists z'.([\![z]\!](y)^\dagger \otimes [\![V]\!](z)^\dagger) \vdash_{\mathcal{C}_v} [\![V]\!](y)^\dagger}\ \exists\text{-l}$$

– $M = \lambda x'.N$: ce cas est prouvé formellement dans la figure 4.1.

– $M = N.N'$: ce cas est prouvé formellement dans la figure 4.2. □

Lemme 4.9 (Correction de la β_v-réduction)**.** *Soient M un λ-terme, V une valeur et y une variable de $\mathcal{V}_e$.*

$$[\![(\lambda x.M)V]\!](y)^\dagger \vdash_{\mathcal{C}_v} \exists z'([\![M[x\backslash z']]\!](y)^\dagger \otimes [\![V]\!](z')^\dagger)$$

Preuve. La preuve formelle de ce séquent est en figure 4.3. Pour cette preuve, on suppose, sans perte de généralité, que les variables x, y, z et z' sont deux à deux distinctes. De plus, par soucis de concision les contraintes *value* et *apply* sont abrégées respectivement en *val* et en *app*. □

Proposition 4.10 (Correction Logique)**.** *Soient M et N deux λ-termes et y une variable de $\mathcal{V}_e$.*

$$\textit{Si } M \xrightarrow{*}_{\beta_v} N \textit{ alors } [\![M]\!](y) \vdash_{\mathcal{C}_v} [\![N]\!](y)$$

Preuve. Ce résultat se démontre par induction sur $M \xrightarrow{*}_{\beta_v} N$. Le cas de base (où $M \xrightarrow{*}_{\beta_v} M$) est immédiat par réflexivité de $\vdash_{\mathcal{C}}$. Dans le cas inductif, quand $M \xrightarrow{*}_{\beta_v} L \rightarrow_{\beta_v} N$, on sait par hypothèse d'induction, que $[\![M]\!](y) \vdash_{\mathcal{C}_v} [\![L]\!](y)$. Il ne reste plus qu'à prouver que $[\![L]\!](y) \vdash_{\mathcal{C}_v} [\![N]\!](y)$, ce qui peut être fait par induction sur L :

– : $L = (\lambda x.L')V$: il suffit d'appliquer les deux lemmes précédents.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{apply(y,x',z)\otimes value(x')\vdash_{\mathcal{C}_v} apply(y,x',z)\otimes value(x')}\ \text{x} \quad \dfrac{}{[\![N[x\backslash z']\!](y)^\dagger\otimes[\![V]\!](z')^\dagger\vdash_{\mathcal{C}_v}[\![N[x\backslash V]\!](y)^\dagger}\ \text{H.I.}}{(apply(y,x',z)\otimes value(x'))\multimap[\![N[x\backslash z']\!](y)\otimes[\![V]\!](z')^\dagger\vdash_{\mathcal{C}_v}(apply(y,x',z)\otimes value(x'))\multimap[\![N[x\backslash V]\!](y)^\dagger}\ \multimap\text{-r},\ \multimap\text{-l}}{!\forall x'z((apply(y,x',z)\otimes value(x'))\multimap[\![N[x\backslash z']\!](y)^\dagger)\otimes[\![V]\!](z')^\dagger\vdash_{\mathcal{C}_v}!\forall x'z((apply(y,x',z)\otimes value(x'))\multimap[\![N[x\backslash V]\!](y)^\dagger)}\ \text{!-p, !-d, } (\forall\text{-r})\times2,\ (\forall\text{-l})\times2}{[\![\lambda x'.N[x\backslash z']\!](y)^\dagger\otimes[\![V]\!](z')^\dagger\vdash_{\mathcal{C}_v}[\![\lambda x'.N[x\backslash V]\!](y)^\dagger}\ [\![\,]\!]}{\exists z'([\![\lambda x'.N[x\backslash z']\!](y)^\dagger)\otimes[\![V]\!](z')^\dagger)\vdash_{\mathcal{C}_v}[\![\lambda x'.N[x\backslash V]\!](y)^\dagger}\ \exists\text{-l}$$

FIG. 4.1 – Preuve du séquent $\exists z'([\![\lambda x'.N[x\backslash z']\!]()^\dagger)\otimes[\![V]\!](z')^\dagger)\vdash_{\mathcal{C}_v}[\![\lambda x'.N[x\backslash V]\!](y)^\dagger$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}[\![V]\!](z)^\dagger\otimes[\![V]\!](z)^\dagger}\ \text{x-5} \quad \dfrac{\dfrac{}{[\![N[x\backslash z]\!](w)^\dagger,[\![V]\!](z)^\dagger\vdash_{\mathcal{C}_v}[\![N[x\backslash V]\!](w)^\dagger}\ \text{H.I.} \quad \dfrac{}{[\![N[x\backslash z]\!](w)^\dagger,[\![V]\!](z)^\dagger\vdash_{\mathcal{C}_v}[\![N[x\backslash V]\!](w)^\dagger}\ \text{H.I.}}{[\![N[x\backslash z]\!](w)^\dagger,[\![N'[x\backslash z]\!](w')^\dagger,[\![V]\!](z)^\dagger,[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}[\![N[x\backslash V]\!](w)^\dagger\otimes[\![N'[x\backslash V]\!](w')^\dagger}\ \otimes\text{-r}}{[\![N[x\backslash z]\!](w)^\dagger,[\![N'[x\backslash z]\!](w')^\dagger,[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}[\![N[x\backslash V]\!](w)^\dagger\otimes[\![N'[x\backslash V]\!](w')^\dagger}\ \text{cut, }\otimes\text{-l}}{apply(w,w',y),[\![N[x\backslash z]\!](w)^\dagger,[\![N'[x\backslash z]\!](w')^\dagger,[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}apply(w,w',y)\otimes[\![N[x\backslash V]\!](w)^\dagger\otimes[\![N'[x\backslash V]\!](w')^\dagger}\ \otimes\text{-r}}{apply(w,w',y)\otimes[\![N[x\backslash z]\!](w)^\dagger\otimes[\![N'[x\backslash z]\!](w')^\dagger\otimes[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}apply(w,w',y)\otimes[\![N[x\backslash V]\!](w)^\dagger\otimes[\![N'[x\backslash V]\!](w')^\dagger}\ (\otimes\text{-l})\times3}{\exists z'(\exists ww'(apply(w,w',y)\otimes[\![N[x\backslash z]\!](w)^\dagger\otimes[\![N'[x\backslash z]\!](w')^\dagger)\otimes[\![V]\!](z)^\dagger)\vdash_{\mathcal{C}}\exists ww'(apply(w,w',y)\otimes[\![N[x\backslash V]\!](w)^\dagger\otimes[\![N'[x\backslash V]\!](w')^\dagger)}\ (\exists\text{-l})\times3,(\exists\text{-r})\times2}{\exists z'[\![N[x\backslash z]N'[x\backslash z]]\!](y)\otimes[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}[\![N[x\backslash V]N'[x\backslash V]]\!](y)^\dagger}\ [\![\,]\!]$$

FIG. 4.2 – Preuve du séquent $\exists z'[\![N[x\backslash z]N'[x\backslash z]]\!](y)\otimes[\![V]\!](z)^\dagger\vdash_{\mathcal{C}}[\![N[x\backslash V]N'[x\backslash V]]\!](y)^\dagger$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![M[x\backslash z']\!](y)^\dagger \vdash_{c_v} [\![M[x\backslash z']\!](y)^\dagger}\,\text{x}}{[\![V]\!](z')^\dagger \vdash_{c_v} [\![V]\!](z)^\dagger}\,\text{x}}{[\![M[x\backslash z']\!](y)^\dagger, [\![V]\!](z')^\dagger \vdash_{c_v} [\![M[x\backslash z']\!](y)^\dagger \otimes [\![V]\!](z)^\dagger}\,\otimes\text{-r}}{[\![M[x\backslash z']\!](y)^\dagger, !val(z), [\![V]\!](z')^\dagger \vdash_{c_v} [\![M[x\backslash z']\!](y)^\dagger \otimes [\![V]\!](z)^\dagger}\,!\text{-a}}{val(z) \vdash_{c_v} !val(z)}\,\text{x-}\tilde{\text{b}}}{[\![M]\!](y)^\dagger[x\backslash z'], val(z), [\![V]\!](z')^\dagger \vdash_{c_v} [\![M[x\backslash z]\!](y)^\dagger \otimes [\![V]\!](z)^\dagger}\,\text{cut}}{[\![V]\!](z')^\dagger \vdash_{c_v} [\![V]\!](z')^\dagger \otimes val(z')\quad (app(z,z',y), val(z'), (\forall xy(app(z,x,y) \otimes val(x) \multimap [\![M]\!](y)^\dagger)), val(z), [\![V]\!](z')^\dagger \vdash_{c_v} [\![M[x\backslash z']\!](y)^\dagger \otimes [\![V]\!](z)^\dagger}\,(\forall\text{-l})\times 2,\ \multimap\text{-l}}{(app(z,z',y), (\forall xy(app(z,x,y) \otimes val(x) \multimap [\![M]\!](y)^\dagger)), val(z), [\![V]\!](z') \vdash_{c_v} [\![M[x\backslash z']\!](y) \otimes [\![V]\!](z)^\dagger}\,\text{cut}}{(app(z,z',y), !(\forall xy(app(z,x,y) \otimes val(x) \multimap [\![M]\!](y)^\dagger)), val(z), [\![V]\!](z')^\dagger \vdash_{c_v} [\![M[x\backslash z']\!](y)^\dagger \otimes [\![V]\!](z)^\dagger}\,!\text{-d}}{\exists zz'(app(z,z',y)\otimes !(\forall xy(app(z,x,y) \otimes val(x) \multimap [\![M]\!](y)^\dagger)) \otimes val(z) \otimes [\![V]\!](z')^\dagger \vdash_{c_v} [\![M[x\backslash z]\!](y)^\dagger \otimes [\![V]\!](z)^\dagger}\,(\otimes\text{-r})\times 3}{\exists zz'(app(z,z',y)\otimes !(\forall xy(app(z,x,y) \otimes val(x) \multimap [\![M]\!](y)^\dagger)) \otimes val(z) \otimes [\![V]\!](z')^\dagger \vdash_{c_v} \exists z([\![M[x\backslash z]\!](y)^\dagger \otimes [\![V]\!](z)^\dagger)}\,\exists\text{-r,}(\exists\text{-l})\times 2}{[\![(\lambda x.M)V]\!](y)^\dagger \vdash_{c_v} \exists z([\![M[x\backslash z]\!](y)^\dagger \otimes [\![V]\!](z)^\dagger)}\,[\![]\!]$$

FIG. 4.3 – Preuve du séquent $[\![(\lambda x.M)V]\!](y) \vdash_{c_v} \exists z([\![M[x\backslash z]\!](y) \otimes [\![V]\!](z))$

– : $L = L'L''$: Dans ce cas on infère que $N = N'N''$ avec $L' \xrightarrow{*}_{\beta_V} N'$ et $L'' \xrightarrow{*}_{\beta_V} N''$. Par hypothèses d'inductionon a donc $[\![L']\!](x')^\dagger \vdash_{\mathcal{C}_v} [\![N']\!](x')^\dagger$ et $[\![L'']\!](x'')^\dagger \vdash_{\mathcal{C}_v} [\![N'']\!](x'')^\dagger$. Pour conclure il suffit de constater que pour toutes formules A_1, A_2, B_1 et B_2 telles que $A_1 \vdash_{\mathcal{C}_v} A_2$ et $A_2 \vdash_{\mathcal{C}_v} A_2$ on a $\exists x'x''.(apply(x', x'', y) \otimes A_1 \otimes B_1) \vdash \exists x'x''.(apply(x', x'', y) \otimes A_2 \otimes B_2)$. □

Théorème 4.11 (Correction)**.** *Soient M et N deux λ-termes et y une variable de $\mathcal{V}_e$. Si $M \xrightarrow{*}_{\beta_V} N$ alors il existe une configuration κ telle que $\langle \emptyset; \mathbf{1}; [\![M]\!](y)\rangle \longrightarrow \kappa$ et $\mathcal{O}^{ps}([\![N]\!](y)) = \mathcal{O}^{ps}(\kappa)$.*

Preuve. Il suffit d'utiliser la proposition précédente et la sémantique logique de LCC. □

4.3.2 Complétude

Pour prouver la complétude, il est nécessaire de faire le lien entre les contraintes de synchronisation présentes dans le store et les positions du terme encodé. Comme l'a illustré l'exemple 4.6, il existe un lien entre les positions d'un terme encodé et les variables d'encodage. Ainsi une contrainte de synchronisation représente une réduction possible dans le λ-terme.

Dans ce chapitre, on notera Λ^+ l'ensemble des λ-terme construit à partir de $\mathcal{V}_\lambda$ et *varse*. On appellera les éléments de Λ^+ λ-termes *étendus*.

Définition 4.12. Soit Γ est un multiensemble de *ask* et c une contrainte. La relation $\gg_{(c;\Gamma)} : \Lambda^+ \times \Lambda$ est définie comme l'ensemble vide si $c \vdash_{\mathcal{C}_v} \exists z_1 z_2 z_3.apply(z_1, z_2, z_3) \otimes \top$ ou sinon comme la plus petite relation satisfaisant les règles suivantes :

$$\frac{}{x \gg_{(c;\Gamma)} x}\ (\mathcal{V}_\lambda) \qquad \frac{z \in \mathcal{V}_e \quad c \vdash_{\mathcal{C}_v} x = z \otimes value(z) \otimes \top}{z \gg_{(c;\Gamma)} x}\ (\mathcal{V}_e\text{-}\mathcal{V}_\lambda)$$

$$\frac{N_1 \gg_{(c;\Gamma)} M_1}{\lambda x.N_1 \gg_{(c;\Gamma)} \lambda x.M_1}\ \textbf{fun} \qquad \frac{N_1 \gg_{(c;\Gamma)} M_1 \quad N_2 \gg_{(c;\Gamma)} M_2}{N_1 N_2 \gg_{(c;\Gamma)} M_1 M_2}\ \textbf{app}$$

$$\frac{\begin{array}{c} N_1 \gg_{(c;\Gamma)} M_1 \quad c \vdash_{\mathcal{C}_v} value(z) \otimes \top \\ \Gamma \equiv \forall x z'(apply(z, x, z') \otimes value(x) \Rightarrow [\![N_1]\!](z')), \Gamma \end{array}}{z \gg_{(c;\Gamma)} \lambda x.M_1}\ (\mathcal{V}_e)\textbf{-fun}$$

On dira que la couple le couple $(c;\Gamma)$ est *cohérent* si pour tout λ-terme M et toutes variables $y \in \mathcal{V}_e$ et $x, x' \in \mathcal{V}_\lambda$ les deux propositions $y \gg_{(c;\Gamma)} \lambda x.M$ et $y \gg_{(c;\Gamma)} x'$ sont mutuellement exclusives.

La relation $\ggg_{(c;\Gamma)} : \Lambda^+ \times \Lambda$ est définie comme l'ensemble vide si $(c;\Gamma)$ n'est pas *cohérent* ou sinon comme la plus petite relation contenant $\gg_{(c;\Gamma)}$ et satisfaisant la règle suivante :

$$\frac{z_1 \ggg_{(e_1;\Gamma)} M_1 \quad z_2 \ggg_{(e_2;\Gamma)} M_2 \quad c \dashv\vdash_{\mathcal{C}_v} apply(z_1, z_2, z) \otimes e_1 \otimes e_2 \quad c \not\vdash_{\mathcal{C}_v} value(z) \otimes \top}{z \ggg_{(c;\Gamma)} M_1 M_2} \ (\mathcal{V}_e)\textbf{-app}$$

Proposition 4.13. *Pour tout λ-terme $M \in \Lambda$, la proposition $M \gg_{(e;\Gamma)} M$ est prouvable.*

Proposition 4.14. *Soit $M \in \Lambda$ un λ-terme, $N \in \Lambda^+$ un λ-terme étendu, $V \in \Lambda$ une valeur et $x \in \mathcal{V}_\lambda$ et $y \in \mathcal{V}_e$ deux variables :*

$$Si \ \ x \notin \mathcal{V}(c)\,,\ N \gg_{(c;\Gamma)} M \ \ et\ y \gg_{(c;\Gamma)} V \quad alors \quad N[x\backslash y] \gg_{(c;\Gamma)} M[x\backslash V]$$

Preuve. Le résultat se démontre par induction sur l'arbre de preuve π de $N \gg_{(c;\Gamma)} M$:

- π finit par la règle $(\mathcal{V}_\lambda)$:

$$\frac{}{x' \gg_{(c;\Gamma)} x'}$$

  Il y a deux sous-cas : $x' \neq x$ ou $x' = x$. Dans le premier cas la règle $(\mathcal{V}_\lambda)$ peut toujours s'appliquer, dans il suffit d'utiliser la preuve de $y \gg_{(c;\Gamma)} V$.

- π finit par la règle $(\mathcal{V}_e\text{-}\mathcal{V}_\lambda)$:

$$\frac{z \in \mathcal{V}_e \quad c \vdash_{\mathcal{C}_v} z = x' \otimes value(z) \otimes \top}{z \gg_{(c;\Gamma)} z'}$$

  Comme précédemment il y a deux sous-cas suivant : $x' \neq x$ ou $x' = x$. Dans le premier cas la règle $(\mathcal{V}_e\text{-}\mathcal{V}_\lambda)$ peut toujours s'appliquer. On remarque que le second cas impossible car il impose $x \in \mathcal{V}(c)$.

- Pour les trois dernière règle il suffit d'utiliser l'hypothèse d'induction.

□

Proposition 4.15. *Soit c une contrainte, Γ un multiensemble de ask persistants et M un λ-terme pure, N un λ-terme étendu et $y \in \mathcal{V}$ une variable fraîche. Si $(c;\Gamma)$ est cohérent et $N \gg_{(c;\Gamma)} M$ alors la prochaine suspension de $\langle X; c; [\![N]\!](y), \Gamma\rangle$ est de la forme $\kappa = \langle Y; c \otimes d; \Gamma, \Delta\rangle$ telle que $y \ggg_{(c\otimes d;\Gamma,\Delta)} M$.*

Preuve. Le résultat se démontre par induction sur l'arbre de preuve π de $N \gg_{(c;\Gamma)} M$:

– π finit par la règle ($\mathcal{V}_\lambda$)ou ($\mathcal{V}_e$-$\mathcal{V}_\lambda$) :

$$\frac{}{z \gg_{(c;\Gamma)} x}\text{avec } z = x \qquad \text{ou} \qquad \frac{z \in \mathcal{V}_e \quad c \vdash_{\mathcal{C}_v} z = x \otimes value(z) \otimes \top}{z \gg_{(c;\Gamma)} x}$$

κ est de la forme $\langle X;\, c \otimes z = y \otimes value(z);\, \Gamma\rangle$. La proposition se prouve donc grâce à la règle ($\mathcal{V}_e$-$\mathcal{V}_\lambda$) et la transitivité de $=$.

– π finit par la règle **fun** :

$$\frac{N_1 \gg_{(c;\Gamma)} M_1}{\lambda x.N_1 \gg_{(c;\Gamma)} \lambda x.M_1}\ \textbf{fun}$$

Dans ce cas κ est de la forme $\langle X;\, c \otimes value(y);\, \Gamma, A\rangle$ ou A est un *ask* persistant de la forme $(\forall xz'(apply(z,x,z') \otimes value(x) \Rightarrow [\![N_1]\!](z')), \Gamma)$.La proposition se prouve donc grâce à la règle ($\mathcal{V}_e$)-**fun**.

– π finit par la règle **app** :

$$\frac{N_1 \gg_{(c;\Gamma)} M_1 \quad N_2 \gg_{(c;\Gamma)} M_2}{N_1 N_2 \gg_{(c;\Gamma)} M_1 M_2}\ \textbf{app}$$

Par hypothèse d'induction, la prochaine suspensions de $\langle \emptyset;\, c;\, [\![N_1]\!](y_1), \Gamma\rangle$ (resp. $\langle \emptyset;\, c;\, [\![N_1]\!](y_1), \Gamma\rangle$) est de la forme $\langle Y_1;\, c \otimes d_1;\, \Delta_2, \Gamma\rangle$ (resp. respectivement de la forme $\langle Y_2;\, c \otimes d_2;\, \Delta_1, \Gamma\rangle$) telle que $y_1 \ggg_{(d_1;\Delta)} M_1$ (resp. telle que $y_2 \ggg_{(d_2;\Delta)} M_2$). Par monotonie, on déduit que κ est de la forme $\langle X_1 \uplus X_2 \uplus \{y_1, y_2\};\, d_1 \otimes d_2 \otimes apply(y_1, y_2, y);\, \Delta_1, \Delta_2, \Gamma\rangle$. On conclut un utilisant la règle **app**, et en remarquant que si $L \gg_{(c;\Gamma)} L'$ alors $L \gg_{(c;\Gamma,\Delta)} L'$. □

Proposition 4.16. *Soient M un λ-terme, y une variable de $\mathcal{V}_e$ et δ une dérivation LCC de la forme :*

$$\delta = (\langle \emptyset;\, \mathbf{1};\, [\![M]\!](y)\rangle \equiv \kappa_0 \xrightarrow{*} \kappa_1 \xrightarrow{*} \kappa_2 \xrightarrow{*} \cdots \xrightarrow{*} \kappa_n)$$

Si chaque κ_i (pour $0 < i \leq n$) est une prochaine suspension de κ_{i-1} alors il existe une dérivation plus générale que δ de la forme :

$$\delta = (\langle \emptyset;\, \mathbf{1};\, [\![M]\!](y)\rangle \equiv \kappa_0 \xrightarrow{*} \langle X_2;\, c_2;\, \Gamma_2\rangle \xrightarrow{*} \cdots \xrightarrow{*} \langle X_1;\, c_1;\, \Gamma_n\rangle)$$

et une dérivation dans le λ-calcul de la forme

$$M \to_{\lambda_V} M_1 \to_{\lambda_V} M_2 \to_{\lambda_V} \cdots \to_{\lambda_V} M_n$$

telle que pour tout i $(0 < i \leq n)$ on a $y \ggg_{(c_1;\Gamma_1)} M_i$.

Preuve. On procède par induction sur n :

- Pour le cas de base, où $n = 0$, on sait, grâce à la proposition 4.13, que $M \gg_{(\mathbf{1};\emptyset)} M$ et $x' \gg_{(\mathbf{1};\emptyset)} x'$. En appliquant la proposition 4.14 avec $x \notin \mathcal{V}(M)$ on déduit que la prochaine suspension de $\langle \emptyset; \mathbf{1}; [\![M]\!](y)\rangle$ est de la forme $\langle X; c_0; \Gamma_0\rangle$ telle que $y \ggg_{(\mathbf{c_0};\Gamma_0)} M$.
- Dans le cas induction, où $n > 0$, on note que la première réduction de κ_n est nécessairement la réduction d'un *ask* de la forme :

$$\forall xz(apply(z_3, x, z') \Rightarrow [\![M]\!](z'))$$

avec $c_n \vdash_{\mathcal{C}_v} value(z_3) \otimes \top$, on déduit alors que $c_n \vdash_{\mathcal{C}_v} apply(z_1, z_2, z_3) \otimes value(z_2)\top$ pour certaines variables z_1 et z_2. L'arbre de preuve π de $y \ggg_{(c_n;\Gamma_n)} N_n$ termine, donc, nécessairement par une règle $(\mathcal{V}_e)$-**app** de la forme :

$$\frac{z'_1 \ggg_{(e_1;\Gamma_n)} M_1 \quad z'_2 \ggg_{(e_2;\Gamma_n)} M_2 \qquad c_n \dashv\vdash_{\mathcal{C}_v} apply(z'_1, z'_2, z'_3) \otimes e_1 \otimes e_2 \quad c_n \nvdash_{\mathcal{C}_v} value(z'_3) \otimes \top}{z \ggg_{(c;\Gamma_n)} M_1 M_2}$$

On procède par induction sur la preuve π.

- $c \vdash_{\mathcal{C}_v} (z'_1, z'_2, z'_3) = (z_1, z_2, z_3) \otimes \top$. Dans ce cas, on déduit que M_1 et M_2 une valeur (en effet la règle $(\mathcal{V}_e)$-**app**est inapplicable pour prouver $z_1 \ggg_{(e_2;\Gamma_n)} M_1$ ou $z_2 \ggg_{(e_2;\Gamma_n)} M_2$, car $c_n \vdash_{\mathcal{C}_v} value(z_1) \otimes value(z_2)\top$). De même grâce à la cohérence de $(c_n; \Gamma_n)$ on infère que M_1 est une fonction de la forme $\lambda x.L$. On conclut à l'aide de la proposition précédente.
- pour le cas $c \nvdash_{\mathcal{C}_v} (z'_1, z'_2, z'_3) = (z_1, z_2, z_3) \otimes \top$, on se contente d'utiliser l'hypothèse d'induction. □

Le théorème suivant est un corollaire de la proposition précédente.

Théorème 4.17 (Complétude)**.** *Soient M un λ-terme, κ une configuration et y une variable de $\mathcal{V}_e$. Si $\langle \emptyset; \mathbf{1}; [\![M]\!](y)\rangle \xrightarrow{*} \kappa$ alors il existe un λ-terme N telle que :$M \xrightarrow{*}_{\beta_V} N$ et $c \in \mathcal{O}^{ps}(\kappa)$ si et seulement si il existe $d \in \mathcal{O}^{ps}([\![N]\!](y))$ tq $d \vdash_{\mathcal{C}_v} c$.*

4.4 Évaluation Paresseuse

On peut distinguer deux grandes familles de langages de programmation fonctionnelle :

- Les langages à évaluation stricte, comme ML[GMM+78], où les arguments sont évalués avant le corps d'une fonction.
- Les langages à évaluation paresseuse, comme Haskell[HJW+92], où les arguments sont évalués à la demande. Les arguments d'une fonction sont passés sous une forme non-évaluée et sont évalués quand, et seulement quand, le calcul en a besoin. De plus, quand un argument est évalué, sa valeur est sauvegardé dans un cache et ne sera jamais plus réévalué.

Comme cela a été dit précédemment, la sémantique de la première classe est définie par la réduction en « appel par valeur » du λ-calcul présentée en section 4.2. Par contre l'évaluation paresseuse ne peut être décrite comme une simple stratégie de réécriture des λ-termes, mais nécessite un codage plus complexe, comme celui proposé par Launchbury dans [Lau93]. Nous proposons un codage de cette stratégie d'évaluation à l'aide d'une traduction proche de la précédente.

Définition 4.18. Chaque terme M du λ-calcul est codé par $[\![M]\!]$ une fonction des variables vers les agents LCC définie récursivement par :

$$[\![x]\!](y) \stackrel{def}{=} (x=y)$$
$$[\![\lambda x.M]\!](y) \stackrel{def}{=} \forall xz(apply(y,x,z) \Rightarrow [\![M]\!](z))$$
$$[\![MN]\!](y) \stackrel{def}{=} apply(z,z',y) \parallel [\![M]\!](z) \parallel needed(z) \parallel needed(z') \rightarrow [\![N]\!](z')$$

où y, z et z' sont supposées appartenir à $\mathcal{V}_e$. Comme précédemment, on suppose que l'ensemble des variables $\mathcal{V}$ est partitionné en deux ensembles dénombrables $\mathcal{V}_\lambda$ et $\mathcal{V}_e$. De plus $\mathcal{C}_p$ est supposé être le système de contraintes construit à partir des axiomes non-logiques suivants :

1. $x=y \Vdash_{\mathcal{C}_p} !x=y$ (= est une contrainte classique)
2. $\Vdash_{\mathcal{C}_p} x=x$ (= est réflexive)
3. $x=y \Vdash_{\mathcal{C}_p} y=x$ (= est symétrique)
4. $x=y \otimes y=z \Vdash_{\mathcal{C}_p} x=z$ (= est transitive)
5. $needed(x) \otimes x=y \Vdash_{\mathcal{C}_p} needed(y)$ (Substitution pour $needed$)
6. $apply(x,y,z) \otimes (x,y,z) = (x',y',z') \Vdash_{\mathcal{C}_p} !apply(x',y',z')$ (Substitution pour $apply$)

La preuve de correction se démontre comme précédemment à l'aide de l'implication logique des traductions. On notera que la présence d'un ask non persistant, nécessaire pour l'encodage d'un terme appliqué (c.-à-d. le sous-terme gauche d'une application) nous fait perdre l'implication logique

«forte» (c.-à-d. $A \vdash_{\mathcal{C}_p} B$) que l'on obtenait pour le codage de la résolution en «appel par valeur». Il faudra se contenter de l'implication logique «faible» (c.-à-d. $A \vdash_{\mathcal{C}_p} B \otimes \top$). La preuve de complétude du codage paresseux se démontre de façon analogue à la version du codage strict : il est nécessaire de faire un lien entre les variables logiques du programme et les sous-terme du λ-terme encodé.

4.5 Discussion

La comparaison des encodages présentés dans ce chapitre avec ceux proposés par Milner [Mil90] pour le π-calcul permet d'aller plus loin dans la comparaison entre variables logique et canaux de communication commencée dans [LM92] et [Sol04]. Tout d'abord notons que dans tous les codages du λ-calcul, le mécanisme de β-réduction connecte le canal représentant l'argument de la fonction réduite avec le canal représentant la variable d'abstraction. En LCC, cette connexion peut se faire simplement par une unification, sans présupposer de la direction dans laquelle circulera l'information à travers ces canaux. Ainsi, en LCC, une variable du λ-calcul est encodée par une unification ($[\![x]\!](y) = (x = y)$), quelque soit la stratégie d'évaluation. En π-calcul, il n'est pas possible de connecter des canaux ensembles sans le présupposé de direction. L'encodage d'une variable dans une stratégie paresseuse où l'information circule de la fonction vers son argument ($[\![x]\!](y) = \bar{x}y$) est donc foncièrement différent de sa version en «appel par valeur» où l'information transite de l'argument vers la fonction ($[\![x]\!](y) = y(z).(!z(w).\bar{x}w)$). On retrouve ici l'avantage de réversibilité des programmes logiques sous la forme de la réversibilité des canaux-variables.

Chapitre 5

Calculs de processus et Langages LCC

Sommaire

5.1 Introduction

À la différence des processus dans les calculs comme le π-calcul [MPW92] qui possède toute l'information nécessaire aux calculs qu'ils décrivent, les agents CC/LCC classiques tels que définis dans [SRP91] et [FRS01], n'ont de sens que s'ils sont considérés avec une information extérieure, les déclarations. Alors qu'un processus du π-calcul décrit une machine de calcul complète (c.-à-d. un état et du code) l'agent LCC classique ne représente en fin de compte qu'un état.

L'internalisation des déclarations aux agents LCC ouvre la voie au traitement théorique des agents LCC comme des processus du π-calcul. Pour illustrer la similitude entre agents et processus, cette section reprend les résultats de Soliman [Sol04]. Dans un second temps, nous proposerons et étudirons une notion d'équivalence observationnelle entre agents plus adaptée au cadre logique que la classique bisimulation. Ce chapitre sera conclut par une discussion sur les liens qu'entretiennent équivalence logique, β-équivalence et équivalence observationnelle.

5.2 Le π-calcul asynchrone

la syntaxe et la sémantique du π-calcul asynchrone, telles que données dans [Bou92] sont rappelées brièvement :

Définition 5.1 (Processus)**.** Soit $\mathcal{N}$ un ensemble infini de *noms*, que nous noterons $x, y, z, \ldots$ La syntaxe des *processus* est la suivante :

$$P ::= \bar{x}z \mid x(y)P \mid P \| P \mid P + P \mid (\nu x)P$$

Intuitivement, $\bar{x}z$ envoie le message z sur le canal x, $x(y)P$ reçoit un message sur le canal nommé x puis exécute le processus P où y représente le message reçu, $P \| P$ exécute deux processus en parallèle, $!P$ duplique un nombre indéfini de fois le processus P et $(\nu x)P$ restreint la portée de x à P.

Définition 5.2 (Relation de Réduction)**.** La sémantique opérationnelle est donnée, comme pour LCC, dans le style de la CHAM [BB92], avec $\equiv$, la *congruence structurelle*, définie comme la plus petite congruence sur les processus satisfaisant les règles :

- $P \| Q \equiv Q \| P$
- $P \| (Q \| R) \equiv (P \| Q) \| R$
- $(\nu x)P \| Q \equiv (\nu x)(P \| Q)$ si $x \notin fn(Q)$
- $(\nu x)P \equiv (\nu y)P[y/x]$ si $y \notin fn(P)$
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
- $(\nu x)P \equiv P$ si $x \notin fn(P)$
- $!P \equiv P \| !P$

La relation de transition $\longrightarrow_\pi$ est la plus petite relation binaire satisfaisant les règles suivantes :

$$x(y)P \| \bar{x}z \longrightarrow_\pi P[z/y] \qquad \frac{P \longrightarrow_\pi P'}{P \| Q \longrightarrow_\pi P' \| Q}$$

$$\frac{P \longrightarrow_\pi P'}{(\nu x)P \longrightarrow_\pi (\nu x)P'} \qquad \frac{P \equiv P', P \longrightarrow_\pi Q, Q \equiv Q'}{P' \longrightarrow_\pi Q'}$$

5.2.1 Encodage du π-calcul asynchrone

Définition 5.3 (Encodage du π-calcul asynchrone)**.** La traduction des processus du π-calcul asynchrone en agents LCC est definie comme :

$$\begin{array}{rcl} [\bar{x}z] & = & msg(x,z) \\ [x(y)P] & = & \forall y(msg(x,y) \rightarrow [P]) \\ [P \parallel Q] & = & [P] \parallel [Q] \\ [!P] & = & \mathbf{1} \Rightarrow [P] \\ [(\nu x)P] & = & \exists x([P]) \end{array}$$

En supposant un système de contraintes vide, c.-à-d. sans axiome logique.

Il peut être noté que la traduction de processus donne des agents qui ne sont pseudo-stables sur aucun sous-langage de $\mathcal{C}$ ($\mathbf{1}$ faisant partie de tout langage de contraintes). Le résultat de la sémantique logique pour les succès ne pourra donc pas être utilisé sur tout processus qui utilise l'opérateur de réplication (!). Ceci n'est cependant pas rédhibitoire, les processus généralement considérés en π-calcul ne terminant généralement pas (voir par exemple [Mil99]). Les autres résultats de la sémantique logique de LCC restent, bien entendu, valides.

On retrouve sans souci les résultats de correction et de complétude de Soliman [Sol04] :

5.2.2 Correction et complétude

Proposition 5.4 (Correction)**.** *Soit P et Q deux processus du π-calcul asynchrone si $P \longrightarrow_\pi Q$ alors $(\emptyset; 1; [P]) \longrightarrow (\emptyset; 1; [Q'])$, avec $Q \equiv Q'$ en utilisant uniquement la règle $!R \equiv !R \mid R$.*

Proposition 5.5 (Complétude)**.** *Soit P et Q deux processus du π-calcul asynchrone, si $(\emptyset; 1; [P]) \longrightarrow (\emptyset; 1; [Q])$ alors $P \longrightarrow_\pi Q$.*

5.3 Équivalence Observationnelle

Afin de définir une notion d'équivalence observationnelle raisonnable la notion de contexte sur les agents LCC est introduite formellement :

Définition 5.6 (Contexte)**.** Un *contexte LCC* est défini par la grammaire suivante :

$$C = [\,] \mid C \parallel C \mid \exists x.C \mid c \mid \forall \vec{x}(c \rightarrow C) \mid \forall \vec{x}(c \Rightarrow C)$$

Les observables définis dans le chapitre précédentcondisent à la notion d'équivalence observationnelle suivante :

Définition 5.7 (Équivalence observationnelle)**.** Deux agents LCC A_1 et A_2 sont *observationnellement équivalents* sur un sous-langage de contraintes $\mathcal{D}$ de $\mathcal{C}$ si pour tout contexte C, $\mathcal{O}^{\mathcal{D}}(C[A_1]) = \mathcal{O}^{\mathcal{D}}(C[A_2])$. Dans ce cas, on notera $A_1 \sim_{\mathcal{D}} A_2$.

Dans cette définition, on appellera le contexte C un *observateur*.

Corollaire 5.8. *Toute équivalence observationnelle est une congruence.*

Exemple 5.9. Soit c, d des contraintes et A un agent LCC.

1. $(c \otimes d) \sim_{\mathcal{C}} (c \parallel d)$
2. $(c \to (d \to A)) \sim_{\mathcal{C}} (c \otimes d) \to A$
3. $(!c \to (d \otimes !c \to A)) \sim_{\mathcal{C}} (!c \to (d \to A))$.
4. $(\mathbf{0} \to A) \sim_{\mathcal{C}} \mathbf{1}$.
5. $(c \to d) \sim_{\mathcal{C}} \mathbf{1}$ si $c \vdash_{\mathcal{C}} d$.
6. $(!c \to (d \otimes !c \to (\mathbf{0} \to A)))) \sim_{\mathcal{C}} (!c \to (d \to \mathbf{1}))$.
7. $\exists x.(c \to A) \sim_{\mathcal{C}} c \to \exists x.A$ si $x \notin \mathcal{V}(c)$

L'équivalence observationnelle entre ces agents sera prouvée formellement plus tard dans ce chapitre (cf. exemples 5.17, 5.19 et 5.20). Le cas 6 présente deux agents similaires obtenus par composition des cas 3 et 4. En effet comme $\sim$ est une congruence si pour tout agent A, $C_1[A] \sim_{\mathcal{C}} C_2[A]$ alors pour tout couple d'agents équivalents A_1 et A_2, $C_1[A_1] \sim_{\mathcal{C}} C_2[A_2]$.

5.3.1 Simplification des observateurs

Cette section montre qu'observer un agent à l'aide de contextes arbitrairement compliqués est équivalent à l'observer à l'aide de contextes plus simples consistant à une simple mise en composition parallèle. Les preuves d'équivalence observationnelle d'agents pourront ainsi être grandement simplifiées.

Remarque 5.10. Soit $\{c_1, \ldots, c_n\}$ un multi-ensemble de contraintes, Γ un multi-ensemble de formules MILL. Si pour toute contrainte d, on a $c_1, \ldots, c_n \vdash_{\mathcal{C}} d$ implique $\Gamma \vdash_{\mathcal{C}} d$ alors $\Gamma \vdash_{\mathcal{C}} (c_1 \otimes \cdots \otimes c_n)$.

Proposition 5.11. *Soit A_1 et A_2 deux formules MILL et $c \in \mathcal{C}$ une contrainte. Si pour tout multiensemble de formules Γ on a $\Gamma, A_1 \vdash_{\mathcal{C}} c$ implique $\Gamma, A_2 \vdash_{\mathcal{C}} c$ alors pour tout multiensemble de formule Γ toute séquence de termes $\vec{s}$ et toute séquence de variables $\vec{x}$ tels que $\vec{x} \cap \mathcal{V}(\Gamma, c) = \emptyset$, $\Gamma, A_1[\vec{x} \backslash \vec{s}] \vdash_{\mathcal{C}} c$ implique $\Gamma, A_2[\vec{x} \backslash \vec{s}] \vdash_{\mathcal{C}} c$.*

Preuve. Dans la dérivation formelle suivante, il faut noter que $\vec{x} \cap \mathcal{V}(\Gamma, c) = \emptyset$ dans l'introduction à gauche de la quantification existentielle.

$$\dfrac{\dfrac{\dfrac{}{A_2[\vec{x}\backslash\vec{s}] \vdash_{\mathcal{C}} A_2[\vec{x}\backslash\vec{s}]}\,\text{X} \quad \dfrac{}{\vdash_{\mathcal{C}} \vec{s}=\vec{s}}\,=}{\dfrac{A_2[\vec{x}\backslash\vec{s}] \vdash_{\mathcal{C}} A_2[\vec{x}\backslash\vec{s}] \otimes \vec{s}=\vec{s}}{A_2[\vec{x}\backslash\vec{s}] \vdash_{\mathcal{C}} \exists\vec{x}(A_2 \otimes \vec{s}=\vec{x})}\,\exists\text{-R}}\,\otimes\text{-R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A_1, \vec{x}=\vec{s} \vdash_{\mathcal{C}} A_1[\vec{x}\backslash\vec{s}]}\,= \quad \Gamma, A_1[\vec{x}\backslash\vec{s}] \vdash_{\mathcal{C}} c}{\Gamma, A_1, \vec{x}=\vec{s} \vdash_{\mathcal{C}} c}\,\text{cut}}{\Gamma, A_2, \vec{x}=\vec{s} \vdash_{\mathcal{C}} c}\,\text{Hypothèses}}\,\otimes\text{-L}}{\Gamma, \exists\vec{x}(A_2 \otimes \vec{x}=\vec{s}) \vdash_{\mathcal{C}} c}\,\exists\text{-L}}{\Gamma, A_2[\vec{x}\backslash\vec{s}] \vdash_{\mathcal{C}} c}\,\text{cut}$$

□

Lemme 5.12. *Soit A_1 et A_2 deux traductions d'agents LCC(C). Si pour tout multi-ensemble de formules Γ et toute contrainte c, $\Gamma, A_1 \vdash_{\mathcal{C}} c$ implique $\Gamma, A_2 \vdash_{\mathcal{C}} c$ alors pour tout multi-ensemble de formules Γ, tout contexte C et toute contrainte $c \in \mathcal{C}$, $\Gamma, C[A_1] \vdash_{\mathcal{C}} c$ implique $\Gamma, C[A_2] \vdash_{\mathcal{C}} c$*

Preuve. On procède par induction sur le contexte C (induction principale). Les deux cas de bases, où C est un trou ou un contexte sans trou sont immédiats. La preuve du cas inductif se fait par induction sur la preuve π de $\Gamma, C[A_1] \vdash_{\mathcal{C}} c$ (induction secondaire). Grace à la proposition 3.35, on sait que cette preuve à un sens. On ne s'intéresse, ici, qu'aux cas qui ne sont pas des conséquences immédiates des hypothèses des deux inductions :

- π finit par *axiome* ou par une *coupure* de la forme :

$$\dfrac{}{C[A_1] \vdash_{\mathcal{C}} c} \qquad \text{ou} \qquad \dfrac{\dfrac{}{C[A_1] \vdash_{\mathcal{C}} c'} \quad \Gamma, c' \vdash_{\mathcal{C}} c}{\Gamma, C[A_1] \vdash_{\mathcal{C}} c}$$

  Dans ces deux cas $C[A]$ est de contrainte. On conclut en utilisant la remarque 5.10.

- π finit par une règle d'introduction du $\multimap$ à gauche de la forme :

$$\dfrac{\Delta \vdash_{\mathcal{C}} d \quad \Gamma, C[A_1] \vdash_{\mathcal{C}} c}{\Delta, \Gamma, d \multimap C[A_1] \vdash_{\mathcal{C}} c}$$

  Dans ce cas il suffit d'appliquer l'hypothèse d'induction principale sur la prémisse de droite. 5.10.

- π finit par une règle d'introduction du $\forall$ à gauche de la forme :

$$\dfrac{\Gamma, (C'[A_1])[\vec{t}/\vec{x}] \vdash_{\mathcal{C}} c}{\Gamma, \forall\vec{x}.C'[A_1] \vdash_{\mathcal{C}} c}$$

  Dans ce cas on suppose, sans perte de généralité, que $\vec{x} \cap \mathcal{V}(\Gamma, c) = \emptyset$ et on utilise la proposition 5.11.

– π finit par des *contractions*, on considère la dernière règle $(*)$ qui ne soit pas une contraction. π finit donc par une succession de règle de la forme :

$$\dfrac{\dfrac{}{\Delta, !C'[A_1], \ldots, !C'[A_1] \vdash_{\mathcal{C}} c}\ (*)}{\vdots}$$
$$\dfrac{\Gamma, !C'[A_1], !C'[A_1] \vdash_{\mathcal{C}} c}{\Gamma, !C'[A_1] \vdash_{\mathcal{C}} c}$$

On procède par cas sur la règle $(*)$. On notera que toute les contractions sous $(*)$ peuvent être trivialement permutées les unes par rapport aux autres :

– $(*)$ est une *coupure* ou une introduction à gauche du $\multimap$ ou une introduction à droite du $\otimes$ de la forme :

$$\dfrac{\Delta_1, !C[A_1], \ldots, !C[A_1] \vdash_{\mathcal{C}} d_1 \quad \Delta_2, !C[A_1], \ldots, !C[A_1] \vdash_{\mathcal{C}} d_2}{\Delta, !C'[A_1], \ldots, !C[A_1] \vdash_{\mathcal{C}} c}\ (*)$$

Dans toutes ces cas, toutes les contractions qui dupliquent $!C[A]$ sauf une peuvent être permuter avec $(*)$. On obtient alors un sous-arbre de la forme :

$$\dfrac{\dfrac{\Delta_1, !C[A_1], \ldots, !C[A_1] \vdash_{\mathcal{C}} d_1}{\vdots} \quad \dfrac{\Delta_2, !C[A_1], \ldots, !C[A_1] \vdash_{\mathcal{C}} d_2}{\vdots}}{\dfrac{\Delta_1, !C[A_1], \vdash_{\mathcal{C}} d_1 \qquad \Delta_2, !C[A_1], \vdash_{\mathcal{C}} d_2}{\dfrac{\Delta, !C'[A_1], !C[A_1] \vdash_{\mathcal{C}} c}{\Gamma, !C'[A_1] \vdash_{\mathcal{C}} c}}}\ (*)$$

On conclut par hypothèse d'induction secondaire sur les deux séquents $\Delta_1, !C[A_1], \vdash_{\mathcal{C}} d_1$ et $\Delta_2, !C[A_2], \vdash_{\mathcal{C}} d_2$

– $(*)$ est une *déréliction* de la forme :

$$\dfrac{\Delta, !C[A_1], \ldots, !C[A_1], C[A_1] \vdash_{\mathcal{C}} c}{\Delta, !C[A_1], \ldots, !C[A_1], !C[A_1] \vdash_{\mathcal{C}} c}\ (*)$$

Comme précédemment, les contractions qui dupliquent $!C[A]$ sauf une peuvent être permutées avec $(*)$. On obtient alors un sous-arbre de la forme :

$$\dfrac{\dfrac{\Delta, !C[A_1], \ldots, !C[A_1], C[A_1] \vdash_{\mathcal{C}} c}{\vdots}}{\dfrac{\dfrac{\Delta, !C[A_1], C[A_1] \vdash_{\mathcal{C}} c}{\Delta, !C'[A_1], !C[A_1] \vdash_{\mathcal{C}} c}\ (*)}{\Gamma, !C'[A_1] \vdash_{\mathcal{C}} c}}$$

On applique l'hypothèse d'induction secondaire sur le séquent $\Delta, !C[A_1], C[A_1] \vdash_{\mathcal{C}} c$ pour inférer que $\Delta, !C[A_2], C[A_1] \vdash_{\mathcal{C}} c$. On utilise ensuite l'hypothèse d'induction principale pour conclure.

– Dans tous les autres cas la règle $(*)$ peut être trivialement permutée avec toutes les contractions. On se ramène alors à d'autres cas de l'induction secondaire. □

Lemme 5.13. *Soit A_1 et A_2 deux traductions d'agents LCC($\mathcal{C}$). Si pour tout multiensemble de formules Γ et toute contrainte c, $\Gamma, A_1 \vdash_{\mathcal{C}} c \otimes \top$ implique $\Gamma, A_2 \vdash_{\mathcal{C}} c \otimes \top$ alors pour tout multiensemble de formules Γ, tout contexte C et toute contrainte $c \in \mathcal{C}$, $\Gamma, C[A_1] \vdash_{\mathcal{C}} c \otimes \top$ implique $\Gamma, C[A_2] \vdash_{\mathcal{C}} c \otimes \top$*

Preuve. Cette preuve est sensiblement la même que précédemment. Il suffit de faire une double induction, premièrement sur la taille du contexte C, puis sur la preuve π de $\Gamma, C[A_1] \vdash_{\mathcal{C}} c \otimes \top$. Le seul cas réellement différent est quand π finit par une introduction à droite du $\otimes$. Il y a alors deux sous-cas :

$$\frac{\Delta_1, C[A_1] \vdash_{\mathcal{C}} \top \quad \Delta_2 \vdash_{\mathcal{C}} c}{\Delta_1, \Delta_2, C[A_1] \vdash_{\mathcal{C}} c \otimes \top} \qquad \frac{\Delta_1, C[A_1] \vdash_{\mathcal{C}} c \quad \Delta_2 \vdash_{\mathcal{C}} \top}{\Delta_1, \Delta_2, C[A_1] \vdash_{\mathcal{C}} c \otimes \top}$$

Le premier sous-cas est trivial. Dans le second il suffit d'utiliser le lemme précédent. □

Théorème 5.14. *Soit A_1 et A_2 deux agents LCC($\mathcal{C}$). $A_1 \sim_{\mathcal{C}} A_2$ si et seulement si $\mathcal{O}^{ca}(B \| A_1) = \mathcal{O}^{ca}(B \| A_2)$ pour tout agent B.*

Preuve. Il suffit d'utiliser la sémantique logique de LCC et d'appliquer les deux lemmes précédents. □

Comme le montreront les exemples suivants, ce théorème permet de simplifier grandement les preuves d'équivalence observationnelle.

5.3.2 Simplification des observables

La proposition suivante affirme que l'observation (au sens de la définition de $\sim_{\mathcal{C}}$) sur l'ensemble du langage de contraintes $\mathcal{C}$ peut se ramener à l'observation sur une partie dense du langage $\mathcal{C}$ et vice-versa.

Proposition 5.15. *Soit A_1 et A_2 deux agents, $\mathcal{D}$ un sous-langage dense du langage de $\mathcal{C}$.*

$$A_1 \sim_{\mathcal{D}} A_2 \quad \textit{si et seulement si} \quad A_1 \sim_{\mathcal{D}'} A_2$$

Preuve. La direction « si » est immédiate car $\mathcal{D} \subset \mathcal{C}$. Pour la direction « seulement si » raisonnons par l'absurde : supposons que $A_1 \not\sim_{\mathcal{C}} A_2$. On déduit alors que pour un certain contexte C, il existe une contrainte c qui est accessible pour $C[A_1]$ mais pas pour $C[A_2]$. N'importe qu'elle contrainte $d \in \mathcal{D}$, telle que $C[A_2] \not\vdash_{\mathcal{C}} d \otimes \top$ – nous savons qu'une telle contrainte existe puisque que $\mathcal{D}$ est une partie dense de $\mathcal{C}$ – est donc accessible pour l'agent $(C[A_1] \parallel c \to d)$ mais par pour l'agent $(C[A_2] \parallel c \to d)$. On a alors $A_1 \not\sim_{\mathcal{D}} A_2$. □

5.3.3 Équivalence observationnelle et équivalence logique

Il semble raisonnable de s'interroger sur les liens qui existent entre équivalence observationnelle et équivalence logique.

Proposition 5.16. *Deux agents dont les traductions sont logiquement équivalentes sont observationnellement équivalents.*

Preuve. Immédiat grâce à la sémantique logique de LCC. □

Exemple 5.17. Reprenons certains agents de l'exemple 5.9. Soit c, d des contraintes et A un agent LCC($\mathcal{C}$).

1. $(c \otimes d) \sim_{\mathcal{C}} (c \| d)$
2. $(c \to d \to A) \sim_{\mathcal{C}} (c \otimes d) \to A$
3. $(!c \to (d \otimes !c \to A)) \sim_{\mathcal{C}} (!c \to (d \to A))$.

Les agents du couple 1 admettent la même traduction logique et sont donc observationnellement équivalents. De même comme le montre le lemme C.1 (resp. le lemme C.2) les agents du couple 2 (resp. du couple 3) admettent des traductions logiquement équivalentes et sont donc, eux aussi, observationnellement équivalents.

Néanmoins l'équivalence logique n'est pas une condition nécessaire. En effet, le cas 4 de l'exemple 5.9 ($\mathbf{0} \multimap A \sim_{\mathcal{C}} \mathbf{1}$) présente deux agents observationnellement équivalents mais non logiquement équivalents (car $\mathbf{0} \multimap A \not\vdash_{\mathcal{C}} 1$). En pratique la proposition suivante montre que l'on peut se contenter de l'équivalence logique, appelée ici « faible », $A \otimes \top \dashv\vdash_{\mathcal{C}} B \otimes \top$.

Proposition 5.18. *Soit A_1 et A_2 deux agents LCC(C). Si $(A_1^\dagger \otimes \top) \dashv\vdash_{\mathcal{C}} (A_2^\dagger \otimes \top)$ alors $A_1 \sim A_2$.*

Preuve. Soit A_1, A_2, B et C quatre formules MILL telles que $A_1 \otimes \top \vdash_{\mathcal{C}} A_2 \otimes \top$ et $B \otimes A_2 \vdash_{\mathcal{C}} C \otimes \top$. La preuve formelle suivante montre que sous ces

hypothèses, le séquent $B \otimes A_1 \vdash_{\mathcal{C}} C \otimes \top$ est prouvable. Notons simplemenent que $A_1 \otimes \top \vdash_{\mathcal{C}} A_2 \otimes \top$ est équivalent à $A_1 \vdash_{\mathcal{C}} A_2 \otimes \top$.

$$\dfrac{A_1 \vdash_{\mathcal{C}} A_2 \otimes \top \qquad \dfrac{\dfrac{B \otimes A_2 \vdash_{\mathcal{C}} C \otimes \top \quad \dfrac{}{\top \vdash_{\mathcal{C}} \top}\,\top}{B \otimes A_2, \top \vdash_{\mathcal{C}} C \otimes \top \otimes \top}\,\otimes\text{-R} \quad \dfrac{\dfrac{}{c \vdash_{\mathcal{C}} c}\,\mathrm{X} \quad \dfrac{}{\top, \top \vdash_{\mathcal{C}} \top}\,\top}{C, \top, \top \vdash_{\mathcal{C}} C \otimes \top}\,\text{cut}}{C \otimes \top \otimes \top \vdash_{\mathcal{C}} C \otimes \top}\,\otimes\text{-L}}{B \otimes A_2, \top \vdash_{\mathcal{C}} C \otimes \top}\,\text{cut}}{B, A_2 \otimes \top \vdash_{\mathcal{C}} C \otimes \top}\,\otimes\text{-L}}{B, A_1 \vdash_{\mathcal{C}} C \otimes \top}\,\text{cut}}{B \otimes A_1 \vdash_{\mathcal{C}} C \otimes \top}\,\otimes\text{-L}$$

On conclut en utilisant la sémantique logique de LCC et le théorème 5.14. □

Exemple 5.19. Soit les agents présentés dans les cas 4 et 5 de l'exemple 5.9 :

4. $(\mathbf{0} \to A) \sim \mathbf{1}$.
5. $(c \to d) \sim \mathbf{1}$ Si $c \vdash_{\mathcal{C}} d$.

La dernière proposition permet de déduire facilement que ces deux équivalences observationnelles sont correctes. En effet, il est assez facile de démontrer que $(\mathbf{0} \multimap A) \otimes \top \vdash \mathbf{1} \otimes \top$ (cf. lemma C.3) et que si $c \vdash_{\mathcal{C}} d$ alors $(c \multimap d) \otimes \top \dashv\vdash_{\mathcal{C}} \mathbf{1} \otimes \top$ (cf. lemma C.4).

L'équivalence logique « faible » est donc une condition suffisante pour obtenir l'équivalence observationnelle, elle n'est cependant pas nécessaire, comme le montre l'exemple suivant.

Exemple 5.20. Soit le cas 7 de l'exemple 5.9 : $\exists x.(c \to A) \sim_{\mathcal{C}} c \to \exists x.A$ si $x \notin \mathcal{V}(c)$. Ces deux agents sont clairement observationnellement équivalents, mais ne sont pas logiquement équivalents. En effet, une énumération exhaustive de toutes les règles de déduction applicables, sauf la *coupure*, montre qu'il n'existe pas de preuve en MILL du séquent $(c \multimap \exists x.A) \otimes \top \vdash \exists x.(c \multimap A) \otimes \top$.

5.3.4 Équivalence observationnelle et bisimulation

L'équivalence observationnelle n'est pas une bisimulation. Prenons, par exemple, les deux agents suivants (cas 2 de l'exemple 5.9) :

$$A_1 = (c \to (d \to \mathbf{1})) \qquad A_2 = ((c \otimes d) \to \mathbf{1})$$

Il a été démontré précédemment que ces deux agents étaient observationnellement équivalents (car équivalents logiquement), ils ne sont cependant pas bisimilaires. En effet $\langle \emptyset;\, c;\, A_1 \rangle \longrightarrow \langle \emptyset;\, \mathbf{1};\, (d \to \mathbf{1}) \rangle$ alors que $\langle \emptyset;\, c;\, A_2 \rangle \not\longrightarrow$ et $\langle \emptyset;\, \mathbf{1};\, (d \to \mathbf{1}) \rangle \not\sim \langle \emptyset;\, c;\, A_2 \rangle$.

5.3.5 Équivalence observationnelle et β-équivalence

La notion d'équivalence la plus utilisée dans le λ-calcul est la β-équivalence : Deux λ-termes sont β-équivalents s'ils peuvent se réduire tous les deux vers un même terme. D'un point de vue pratique, on peut donc dire que deux fonctions β-équivalentes calculent la même chose. Le λ-calcul étant un modèle général de calcul, il parait donc judicieux de faire un lien formellement entre une équivalence observationnelle et β-équivalence grâce au codage du λ-calcul « appel par valeur » défini dans la première partie de ce chapitre.

Grâce à la correction logique (proposition 4.10) du codage du λ-calcul « appel par valeur », il est possible de prouver que les observables d'un λ-terme contiennent ceux de son réduit.

Proposition 5.21. *Soit M et N deux λ-termes, $\mathcal{D}$ un sous-langage quelconque de $\mathcal{C}$ et y une variable. Si $M \rightarrow_{\beta_v} N$ alors pour tout contexte C on $\mathcal{O}^{\mathcal{D}}(C[[\![M]\!](y)]) \subset \mathcal{O}^{\mathcal{D}}(C[[\![N]\!](y)])$.*

En revanche l'inclusion dans l'autre direction, n'est pas vraie. Par exemple l'agent $B = \forall xyz(apply(x, y, z) \rightarrow mark)$ (où $mark$ est une contrainte de synchronisation du sous-langage $\mathcal{D}$) utilisé en tant qu'observateur, est capable de distinguer une application d'une valeur. En effet pour tous termes M et N et toute valeur V, $mark \in \mathcal{O}^{\mathcal{D}}(B \parallel [\![MN]\!](y))$ alors que $mark \notin \mathcal{O}^{\mathcal{D}}(B \parallel [\![V]\!](y))$. En constatant que le terme $(\lambda x.V)W$, où V et W sont des valeurs, admet pour seul β-réduit la valeur $V[x\backslash W]$, on conclut que, généralement, $\mathcal{O}^{\mathcal{D}}(B \parallel [\![(\lambda x.V)W]\!](y)) \not\subset \mathcal{O}^{\mathcal{D}}(B \parallel [\![V[x\backslash W]]\!](y))$.

5.4 Discussion

Le problème présenté à la fin de chapitre est assez général et n'est pas lié à l'encodage particulier du λ-cacul. Cela provient du fait qu'il n'est pas possible de cacher les appels (en l'occurrence l'appel au *ask* persistant représentant une fonction interne) fait à l'intérieur d'un agent vis-à-vis d'un observateur extérieur. Grâce à l'utilisation du quantificateur $\forall$, un observateur sera en effet capable d'intercepter n'importe quelle contrainte de synchronisation et cela même s'il ne partage aucune variable avec l'agent observé. Dans la seconde partie de cette thèse, nous proposons une solution élégante à ce problème à travers un système de modules pour LCC.

De même la propriété de simplification des observables (proposition 5.15) est intéressante pour simplifier les preuves d'équivalence observationnelle. Néanmoins elle cache le même défaut, à savoir l'impossibilité de masquer des appels internes à un agent. En effet il serait, en réalité, intéressant de

distinguer l'équivalence observationnelle sur tout le système de contraintes de l'équivalence observationnelle sur la partie classique $\mathcal{D}$ de ce système. Ainsi deux agents qui calculent la même chose, c.-à-d. qui ont le même $\mathcal{D}$-succès, par l'intermédiaire d'algorithmes différents, c.-à-d. en faisait des appels à des procédures distinctes et qui pose donc des contraintes de synchronisation différentes, pourraient être formellement équivalents.

Finalement, bien que l'on ait démontré formellement que l'équivalence logique était distincte de l'équivalence observationnelle, ces deux notions restent très proches. En effet, bien que cela n'est pas été formellement démontrer, les seuls agents équivalents observationnellement, mais non logiquement que l'on ait réussi à exhiber sont des agents dont une interprétation en logique classique sera prouvable. Par exemple, une fois transposée en logique classique les traductions des agents de l'exemple 5.20 sont équivalents : $\exists x.(C \supset A) \dashv\vdash C \supset \exists x.A$ si $x \notin \mathcal{V}(C)$ [1].

Dans la suite de ce mémoire, nous montrons que la définition d'un système de modules simple pour LCC permet de résoudre ces problèmes.

[1] Cette preuve utilise la non-linéarité et le tiers exclu de la logique classique, c.-à-d. que ses équivalents en logique classique intuitionniste ou en logique linéaire classique ne sont pas prouvables.

Deuxième partie

Système de Modules pour les Langages à Variables Logiques

Chapitre 6

Systèmes de Modules pour la PLC

Sommaire

6.1 Introduction

Avant de s'intéresser aux questions de modularité dans le contexte général des langages LCC, nous nous restreignons dans ce chapitre au cadre PLC

en reprennant les résultats publiés dans [HF06]. Cette étude préliminaire a un double intérêt. Tout d'abord, elle permet de nous pencher sur la question d'état de l'art, les questions de modularité n'ayant jamais été étudiées dans le cadre des langages LCC. De plus la sémantique des langages PLC étant plus simple que celle de la classe LCC, il nous sera plus facile d'introduire formellement des concepts tels que la protection du code. Ce chapitre se décompose en trois parties, une partie introductive qui présente brièvement les travaux antérieurs et introduit informellement le concept de protection du code. Dans une seconde partie, nous proposons une étude plus approfondie des principaux systèmes PLC utilisés aujourd'hui. Finalement nous proposons un système de module simple pour PLC et introduisons une définition formelle du concept de protection du code.

6.1.1 Travaux antérieurs

La modularité dans le contexte de la programmation de logique a été considérablement étudiée. Malheureusement il n'existe toujours pas de système de modules consensuel pour Prolog, limitant ainsi le développement de bibliothèques génériques. L'une des difficultés à l'ajout d'un système de module aux langages logiques vient de l'utilisation du prédicat *call* (présenté en section 2.3.3) qui interfère avec les mécanismes de protection du code, une des tâches essentielles que doit réaliser tout système de modules moderne. Il existe de nombreuses propositions de systèmes de modules chacune étant basée sur un compromis entre protection du code et facilité d'utilisation de la méta-programmation.

Il est possible de distinguer différentes approches dans les travaux qui ont déjà été menés à propos de la modularité dans les langages de programmation logique avec contraintes.

Les approches dites *syntaxiques* s'intéressent principalement à l'alphabet des symboles, qu'elles divisent en espace de noms afin de partitionner de grands programmes, de réutiliser sans risque de conflits du code existant et de développer des bibliothèques à usage générique. Ces approches sont souvent utilisées pour des raisons de simplicité et de compatibilité avec les bibliothèques préexistantes. Par exemple des bibliothèques comme le solveur de contraintes linéaires développé par Holzbaur à l'OEAFI [Hol95] ou l'implémentation Prolog du langage CHR (présenté à la sous-section 3.4.3) de Schrijvers [SW04] doivent être portables dans un système de modules pour PLC. La plupart des systèmes de Prolog modulaires, tels que [Swe04], SWI [Wie04], ECLiPSe [Aa01], XSB [Sa03], Ciao [BGC+04, CH00, Cab04] tombent dans cette catégorie.

Les approches dites *algébriques* [O'K85, BLM94, SW92] définissent des calculs de modules avec des opérations pour manipuler des ensembles de clauses. Elles sont dans un certain sens plus générales que les extensions syntaxiques car elles fournissent généralement une grande variété d'opérations sur les modules comme le masquage, la fusion, la surcharge, etc. La grande flexibilité permise par ces approches ce fait au de détriment de des nombreuses techniques de raisonnement sur les programmes, telles que la sémantique logique qui ne peut plus être appliquée sur des programmes modulaires.

Les approches dites *logiques* étendent la logique sous-jacente. On peut citer les extensions avec implications imbriquées [Mil89, Mil94], méta-logique [BMPT92] ou l'ordre supérieur [Che87]. De tels modules "logiques" peuvent être créés dynamiquement de façon similaire aux *contextes* de la Programmation Logique Contextuelle [MP89, AD03]. Peut-être à cause de leur manque de compatibilité avec les systèmes existants, ces systèmes n'ont pas été réellement adoptés et ne seront pas considérés ici. Ces approches ne nous intéressent pas car elle nécessite une extension de la logique linéaire – qui est déjà un raffinement de la logique classique – qui ne nous semble pas nécessaire.

6.1.2 Protection du code

Afin d'imposer une segmentation appropriée du code, et pour garantir la sémantique des prédicats définis dans une bibliothèque, un système de modules doit empêcher strictement l'exécution de procédures non autorisées par le programmeur. Cela signifie qu'il doit être possible de limiter l'accès au code d'un module (par appels de prédicats, appels dynamiques, assertions dynamiques de clauses, modifications de syntaxe, affectations de variables globales, etc.) depuis un code exterieur au module. Cette propriété est appelée *protection du code*.

Les relations entre modules appelés (typiquement une bibliothèque) et modules appelant (typiquement l'utilisateur d'une bibliothèque) étant asymétrique, la propriété de protection du code peut être décomposée en deux sous-propriétés :

- La *protection du module appelé* qui garantit que seuls les prédicats visibles d'un module peuvent être appelés de l'extérieur.
- La *protection du module appelant* qui garantit que les prédicats du module appelant ne peuvent pas être appelés par le module appelé puisqu'ils ne sont pas visibles.

Cependant l'exemple suivant illustre que le second
qu'il est nécessaire de faire une entorse à cette règle.

Exemple 6.1. Soit l'itérateur `forall/2` défini dans l'exemple 2.15.

```
forall([], _).
forall([H|T], P):-  G-..[P,H], call(G),
                    forall(T, P).
```

Un tel prédicat ne peut être défini dans une bibliothèque (typiquement dans la bibliothèque `list`) sans violer la protection du module appelant. En effet, pour avoir le comportement attendu, `forall` doit interpréter le prédicat passé en second argument dans l'environnement du module appelant.

Comme nous le verrons plus tard, les systèmes PLC résolvent cette difficulté soit en abandonnant toute forme de protection du code soit en introduisant des mécanismes *ad hoc* permettant de s'émanciper de certaines contraintes induites par la protection du code.

Dans ce chapitre, nous proposons de garder une protection du code stricte, mais de distinguer les appels dynamiques et les applications de *fermetures*. D'un point de vue fonctionnelle, une fermeture est une lambda expression, c.-à-d. que la nouvelle structure `closure(X,G,C)` est équivalente à `C = (`λ `X.G)` et `apply(C,T)` à `CT`. Cela rend possible le développement d'un module `list` exportant un prédicat `forall` qui prend en argument une fermeture.

Exemple 6.2.

```
:-module(list, [forall/2, ...]).

forall([],_).
forall([H|T],C):- apply(C,H), forall(T,C).
```

Cette définition de `forall` utilisant les fermetures en lieu et place des méta-appels peut être utilisée depuis n'importe quel module important `list`. Par exemple en passant une fermeture du prédicat unaire ISO `var` il est possible d'écrire simplement un prédicat `all_variables` vérifiant que tous les éléments d'une liste sont des variables.

```
:-module(foo, ...).
:-use_module(list).

all_variables(L):- closure(X, var(X), C), forall(L, C).
```

6.2 Aperçu des Systèmes Existants

Dans cette section, nous analysons les principaux systèmes de modules syntaxiques développés pour Prolog. A chaque fois un exemple de référence

sera proposé pour illustrer leurs particularités et les classifier selon les deux propriétés introduites précédemment, à savoir la protection du module appelé et la protection du module appelant.

En suivant la terminologie ISO Prolog [Int00], un *module* est un ensemble de clauses Prolog associé à un identifiant unique, son *nom*. Le *contexte d'appel* – ou simplement *contexte* – est le nom du module à partir duquel un appel est fait. Un *but qualifié* `M:G` est un but Prolog classique `G` préfixé par un nom de module `M` dans lequel il doit être interprété. Un prédicat est *visible* à partir d'un contexte s'il peut être appelé à partir de ce contexte particulier sans qualification. Un prédicat est *accessible* à partir d'un contexte s'il peut être appelé à partir de ce contexte particulier avec ou sans l'utilisation de qualifications. Il est possible d'*importer* dans un module des prédicats définis et déclarés *exportés* dans un autre module, c.-à-d. rendre visible des prédicats d'un autre module. Un *meta-prédicat* est un prédicat qui manipule un ou plusieurs arguments qui doivent être interprétés comme des buts ; ces arguments étant eux-mêmes appelés *méta-arguments*.

6.2.1 Un système de modules basique

Nous considérons d'abord un système de modules basique. Dans ce système, les prédicats visibles à partir d'un module sont soit les prédicats définis dans le module, soit les prédicats importés à partir d'autres modules. Pour garantir la protection du code, seuls les prédicats exportés par un module peuvent être importés dans un autre. De plus la qualification de but est prohibée. Ce système basique garantit trivialement les deux formes de protection du code, mais n'est pas satisfaisant car il interdit l'utilisation de tout itérateur tel que le `forall` défini dans l'exemple 6.1 avec un prédicat non publique.

Il est intéressant de noter que l'étude de Sannella et Wallen [SW92], qui est une des rares approches non syntaxique qui s'intéressent explicitement au traitement des méta-appels, utilise une restriction similaire – c.-à-d. seuls les prédicats déclarés dans la signature d'un module peuvent être utilisés dans un `call` – pour contourner les problèmes de protection induit par les méta-appels.

6.2.2 SICStus Prolog

Le système SICStus Prolog [Swe04] rend accessible tout prédicat, par l'intermédiaire de la qualification de but. L'itérateur `forall` peut être ainsi facilement modularisé en lui passant un but qualifié comme méta-donnée.

En contrepartie de sa souplesse, ce système ne garantit pas la protection du module appelé.

En outre, il est aussi possible, à l'aide de la directive `meta_predicate`, de déclarer explicitement les méta-prédicats et leur méta-données. Dans ce cas, les méta-arguments non qualifiés sont dynamiquement qualifiés par le contexte d'appel du méta-prédicat. Avec un tel comportement, le module appelé peut manipuler explicitement le nom du module d'appel et peut ainsi appeler n'importe quel prédicat dans le module appelant.

Exemple 6.3. Cet exemple, qui sera aussi utilisé dans la suite, exhibe les possibilités d'appeler des prédicats privés (ou non exportés en suivant la terminologie ISO) à partir de modules tiers.

```
                                    :- module(using, [test/0]).
:-module(library, [mycall/1]).     :- use_module(library).

p :- write('library:p/0  ').        p :- write('using:p/0  ').
                                    q :- write('using:q/0  ').
:-meta_predicate(mycall(:)).
mycall(M:G):- M:p, call(M:G).       test :- library:p, mycall(q).
```

Dans ce programme, le prédicat privé `p` de la bibliothèque `library` (module appelé) est appelé à partir du module `using` (module appelant). La bibliothèque, appelle correctement le prédicat `q` (passé comme méta-argument) du module appelant, mais est aussi capable d'appeler le prédicat privé `p` du module appelant.

Ce système de module est très similaire à celui de Quintus Prolog [Swe03] et de Yap Prolog [VSCA00]. De même la proposition de standardisation de Prolog [Int00] est très proche de l'esprit des modules de SICStus, mais les règles d'accessibilité pour les prédicats qualifiés sont laissées à l'implémentation, ce qui rend par la même ce standard peu utile.

6.2.3 ECLiPSe

ECLiPSe [Aa01] introduit deux mécanismes pour les appels de prédicats non visibles. Le premier est l'appel de buts qualifiés, où seuls les prédicats exportés sont visibles. Le second, qui utilise la construction `call(Goal)@Module`, rend accessible n'importe quel prédicat accessible comme peut le faire le but `Module:Goal` dans le système SICStus. De plus le système fournit la directive `tool/2` qui ajoute le contexte d'appel comme argument supplémentaire à un méta-prédicat. Cette solution à l'avantage de limiter les appels aux prédicats

protégés faits de façon non intentionnelle, mais comme précédemment il ne fournit en fin de compte aucune protection du code.

Exemple 6.4.

```
                                  :- module(using, [test/0]).
:- module(library, [mycall/1]).  :- use_module(library).

p :- write('library:p/0  ').      p :- write('using:p/0  ').
                                  q :- write('using:q/0  ').
:- tool(mycall/1,mycall/2).
mycall(G, M):-                    test :-
    call(p)@M, call(G)@M.             call(p)@library, mycall(q).
```

6.2.4 SWI Prolog

Pour des raisons de compatibilité, le système SWI Prolog [Wie04] accepte les buts qualifiés et utilise la même politique que SICStus Prolog pour la gestion de leurs appels. Ainsi ce système ne respecte pas la protection du module appelé. Dans SWI, la méta-programmation utilise une sémantique différente de celle de SICStus. Le contexte d'appel des appels dynamiques, faits dans le corps de clauses déclarées à l'aide de la directive `module_transparent/1`, est le contexte d'appel du but qui a invoqué le méta-prédicat. Ainsi en déclarant l'itérateur `forall/2` comme un prédicat *module transparent*, on obtient le comportement attendu pour l'exemple 6.1, puisque le but `G` est interprété dans le module qui à appelé `forall`, en l'occurrence le module appelant.

Néanmoins, ce choix de comportement a deux inconvénients. Premièrement les appels dynamiques (tels que `call(p(X))`) n'ont pas le même comportement que les appels statiques (tels que `p(X)`, puisque qu'ils ne sont, a priori, pas interprétés dans le même contexte. De plus comme l'illustre l'exemple suivant, cette convention casse la protection du code appelant.

Exemple 6.5.

```
:-module(library, [mycall/1]).   :- module(using, [test/0]).
                                  :- use_module(library).
p :- write('library:p/0  ').
                                  p :- write('using:p/0  ').
:-module_transparent(mycall/1).   q :- write('using:q/0  ').
mycall(G):-
    p, call(p), call(G).          test :- mycall(q).
```

6.2.5 Logtalk

Logtalk [Mou, Mou03][1], n'est pas à proprement parlé un système de modules syntaxique, mais une extension orientée objet de Prolog. Néanmoins en restreignant sa syntaxe – en interdisant notamment la paramétrisation d'objets et l'héritage – on obtient un système de modules très proche de ceux étudiés ici.

La directive `object/1` peut être lue comme la directive `module/2`, où les prédicats publics sont les prédicats exportés. L'envoi de messages joue alors le rôle d'appel qualifié. En effet l'envoi du message `P` à l'objet `M` – qui est dénoté par `M::P` au lieu de `M:P` – appelle le prédicat `P` défini dans l'objet `M` seulement si celui-ci a été déclaré public. Avec un tel comportement, le système respecte la protection du code appelé.

Pour manipuler les méta-prédicats, Logtalk fournit sa propre version de la directive `meta_prédicat`, qui utilise une syntaxe proche de celle de SICStus – `::` est utilisé à la place de `:` pour déclarer les méta-arguments. Comme SWI, Logtalk ne réalise pas l'expansion des méta-arguments (c.-à-d. l'ajout dynamique du contexte d'appel) mais réalise les appels dynamiques dans un contexte qui peut être différent de celui d'un appel statique. Plus précisément, dans ce système, les méta-appels sont interprétés dans le module (c.-à-d. l'objet en utilisant la terminologie Logtalk) qui a envoyé le dernier message. Puisque les buts non-qualifiés ne sont pas considérés comme des envois de messages, le prédicat `call`, peut appeler n'importe quel prédicat du module appelant. Ainsi ce système ne respecte pas la protection du module appelant.

Exemple 6.6.

```
:- object(library).
:- public(mycall/1).

p :- write('library:p/0  ').

mycall(G) :- mycall(G,p).
:-metapredicate(mycall(::,::)).
mycall(G1,G2) :-
    call(G1), call(G2).

:-end_object.
```

```
:- object(using).
:- public(test/0).

p :- write('using:p/0  ').
q :- write('using:q/0  ').

test :- library::mycall(q).

:- end_object.
```

[1] Suite à la publication de notre article [HF06] les défauts présentés dans cette sous-section ont été en partie corrigés.

6.2.6 Ciao Prolog

Le système de modules de Ciao Prolog [BGC+04] respecte les deux formes de protection du code. Seuls les prédicats exportés sont accessibles de l'extérieur d'un module. La manipulation de méta-données à travers le système de modules est rendue possible grâce à une version avancée de la directive `meta_prédicate/1`. Avant un appel à un méta-prédicat, le système traduit dynamiquement un méta-argument en une représentation interne contenant le but et le contexte dans lequel ce dernier doit être appelé. Puisque que cette traduction est faite avant l'appel effectif au méta-prédicat le système est capable de connaître le contexte dans lequel la méta-donnée doit être interprétée. Comme le système ne fournit aucune documentation, ni de prédicat pour manipuler cette donnée interne, il préserve la protection du module appelant. Dans un certain sens, Ciao Prolog fait une distinction entre les termes et les données d'ordre supérieur (c.-à-d. les buts manipulés comme des termes) [CHL04].

Dans l'exemple suivant le programme réalise le comportement attendu sans compromettre aucune des deux propriétés de protection du code.

Exemple 6.7.

```
:-module(library, [mycall/1]).

p :- write('library:p/0  ').

:-meta_predicate(mycall(:)).
mycall(G):-
  writeq(G),write(' '),call(G).
```

```
:- module(using, [test/0]).
:- use_module(library).

p :- write('using:p/0  ').

test :- mycall(p).
```

6.2.7 XSB

Le système XSB [Sa03], qui est lui aussi considéré comme un système syntaxique, suit une approche assez différente des systèmes précédents. En effet, il fait partie de la classe des systèmes *fondés sur les atomes*, au contraire des précédents qui sont eux *fondés sur les prédicats*. La principale différence est que XSB modularise aussi les symboles de fonction. Ainsi deux termes construits dans deux modules différents ne pourront pas, par défaut, être unifiés. Il est toutefois possible d'importer, dans un module, des symboles exportés par un autre module, le système considérant alors que les modules partagent ces symboles. La sémantique du `call` est alors très simple, le méta-appel d'un terme correspond à l'appel au prédicat de même symbole et même

arité, du module où le terme a été créé. Le système respecte ainsi les deux propriétés de protection du code.

Exemple 6.8.

```
:-export mycall/1.

p(_) :- write('library:p/1  ').

mycall(G):- call(G).
```

```
:-export test/1.
:-import mycall/1 from library.

p(_) :- write('using:p/0  ').
test(_) :- mycall(p(_)).
```

Cette solution ne fait cependant que déplacer le problème sur la construction dynamique de termes. En effet, dans XSB les termes construits dynamiquement à l'aide de `=../2`, `functor/3` et `read/1` sont supposés appartenir au module spécial `user` (le module *top-level*). Le système ne respecte donc pas l'indépendance de la stratégie de sélection (par exemple `functor(X, toto, 1), X = toto(_)` échoue alors que `X = toto(_), functor(X, toto, 1)` réussit).

6.3 Un Système de Modules Sûr avec Méta-appels et Fermetures

Dans cette section, nous définissons un système de modules sûr avec méta-appels et fermetures. Nous présentons sa sémantique opérationnelle et démontrons formellement qu'il respecte les deux formes de protections du code présentées précédemment.

Dans ce chapitre, nous reprenons des conventions proches de celles de la section 2.2 pour la définition de PLC non-modulaire, à savoir :

- $\mathcal{V}$ est un ensemble dénombrable de variables dénotées par x, y, ... ;
- Σ_F est un ensemble de symboles de constante ou de fonctions ;
- Σ_C est un ensemble de symboles de prédicat, contenant $true/0$ et $= /2.$;
- Σ_P est un ensemble de symboles de prédicat, contenant $call/2$, $apply/2$ et $closure/3$;
- Σ_M est un ensemble de noms de module, dénotés par μ, ν, ...

De plus, pour interpréter les arguments de *call*, deux relations de coercition $\overset{P}{\rightsquigarrow}: \Sigma_F \times \Sigma_p$ et $\overset{P}{\rightsquigarrow}: \Sigma_M \times \Sigma_p$, permettant d'interpréter les symboles de fonction respectivement comme des symboles de prédicat et comme des noms

de module, sont supposées être définies. Il peut être noté, que dans les systèmes Prolog classiques, où les symboles de fonction ne sont pas distingués de symboles de prédicats, ces relations sont juste l'identité.

6.3.1 Syntaxe des programmes modulaires

Dans cette section, comme dans le reste de ce manuscrit nous emploierons la terminologie, à notre avis plus appropriée, dans le cadre d'un système respectant la protection du code, de *prédicat public* en lieu et place de celle de *prédicat exporté*. Un prédicat non déclaré public sera alors naturellement supposé être *privé*.

Pour des raisons de simplicité, nous adoptons ici les conventions suivantes. Premièrement seule une forme simple de fermeture n'admettant qu'un seul argument sera considérée. Deuxièmement, une syntaxe pour les programmes PLC, qui distingue contraintes, fermetures et atomes dans un but sera employée. Troisièmement, un but est toujours supposé être qualifié, ainsi il ne sera pas nécessaire de décrire la convention standard pour préfixer automatiquement les atomes non qualifiés dans une clause ou un but. Dernièrement tous les prédicats *publics* d'un module seront supposés accessibles de l'extérieur sans prendre en considération une directive telle que `use_module`.

Définition 6.9 (Fermeture)**.** Une *fermeture associée* à une variable z est un atome de la forme $closure(x, \mu\!:\!A, z)$ où x est une variable *d'abstraction*, $\mu\!:\!A$ un atome qualifié. L'*application* d'une fermeture associée à une variable z, à un argument t est l'atome $apply(z, t)$.

Définition 6.10 (Clauses Modulaires)**.** Une *clause modulaire* est une formule de la forme :

$$A_0 \leftarrow c_1, \ldots, c_l \mid \kappa_1, \ldots \kappa_m \mid \mu_1\!:\!A_1, \ldots \mu_n\!:\!A_n.$$

où les c_i sont des contraintes atomiques, les κ_i des fermetures et les $mu_i\!:\!A_i$ des atomes qualifiés.

Définition 6.11 (Modules)**.** Un module est un triplet $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)$ où $\mu \in \Sigma_M$ est le *nom* du module, $\mathcal{D}_\mu$ est un ensemble de clauses appelé *implémentation* du module et $\mathcal{I}_\mu$ est un ensemble de noms de prédicat appelé *interface* du module. Un prédicat p sera dit public dans μ s'il appartient à l'interface $\mathcal{I}_\mu$ de μ ; dans le cas contraire on dira que ce prédicat est *privé*. Un programme *modulaire* est un ensemble de modules dont les noms sont distincts deux à deux.

Définition 6.12 (But)**.** Un *but modulaire* est une formule

$$c \mid \langle \nu_1 - \kappa_1 \rangle, \ldots, \langle \nu_m - \kappa_m \rangle \mid \langle \nu'_1 - \mu_1 : A_1 \rangle, \ldots, \langle \nu'_n - \mu_n : A_n \rangle$$

où c est un ensemble de contraintes atomiques, les κ_i des fermetures, les $\mu_i : A_i$ des atomes qualifiés et les ν_i et ν'_i des noms de modules appelés *contextes d'appel*.

Dans la suite, la construction $\langle \nu - (\kappa_1, \ldots, \kappa_m) \rangle$ sera un raccourci pour la séquence de fermetures avec contexte $\langle \nu - \kappa_1 \rangle, \ldots, \langle \nu - \kappa_m \rangle$. De façon similaire $\langle \nu - (\mu_1 : A_1, \ldots, \mu_n : A_n)) \rangle$ sera un raccourci pour la séquence d'atomes avec contexte $\langle \nu - \mu_1 : A_1 \rangle, \ldots, \langle \nu - \mu_n : A_n \rangle$.

6.3.2 Sémantique opérationnelle

Dans la suite de ce chapitre, $\mathcal{P}$ sera supposé être un programme modulaire défini sur un système de contraintes $\mathcal{X}$ donné.

Définition 6.13 (Relation de transition)**.** La relation de transition $\longrightarrow$ sur les buts modulaires est définie comme la plus petite relation satisfaisant les règles de la table 6.1, où θ est un renommage à l'aide de variables fraîches. Une dérivation *succès* d'un but G est une séquence finie de transition à partir de G et qui termine sur un but ne contenant que des contraintes et des fermetures.

La règle de *CSLD modulaire* est une restriction de la CSLD classique présentée au chapitre 2. La condition supplémentaire $(\nu = \mu) \vee (p \in \mathcal{I}_\mu)$ impose que $\mu : p(\vec{t})$ ne puisse être exécuté que si l'une des deux conditions suivantes est respectée :

- l'appel est fait depuis l'intérieur du module, c.-à-d. depuis le contexte d'appel μ.
- le prédicat p est un prédicat public dans μ.

De plus cette règle impose μ comme nouveau contexte d'appel des fermetures et des atomes introduits par le corps de la clause sélectionnée.

Comme son nom l'indique, la règle *méta-appel* définit la sémantique opération des méta-appels. Il est important de noter que cette règle ne modifie pas le contexte d'appel ν. Cette propriété est essentielle pour garantir la protection du module appelant. Pour des raisons de simplicité, la règle n'est pas capable, telle quelle, de manipuler de conjonctions de buts. Ces méta-appels peuvent néanmoins être facilement émulés en supposant que $(',' \overset{P}{\sim} and/2)$ et en ajoutant la clause $and(x, y) \leftarrow \mu : call(\mu, x), \mu : call(\mu, y)$ à chaque module μ.

CSLD modulaire

$$\frac{\begin{array}{c}(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \qquad (\nu = \mu) \vee (p \in \mathcal{I}_\mu) \\ (p(\vec{s}) \leftarrow c' | k | \beta) \in \mathcal{D}_\mu \qquad \mathcal{X} \models \exists (c \wedge \vec{s}\rho = \vec{t} \wedge c'\rho)\end{array}}{(c|K|\gamma, \langle \nu - \mu : p(\vec{t}) \rangle, \gamma') \longrightarrow (c, \vec{s}\rho = \vec{t}, c'\rho | K, \langle \mu - k\rho \rangle | \gamma, \langle \mu - \beta\rho \rangle, \gamma')}$$

méta-appel

$$\frac{\mathcal{X} \models c \Rightarrow (s = g \wedge t = f(\vec{u})) \qquad g \overset{M}{\sim} \mu \qquad f \overset{P}{\sim} p}{(c|K|\gamma, \langle \nu - \nu : call(s,t) \rangle, \gamma') \longrightarrow (c, s = g, t = f(\vec{x}) | K | \gamma, \langle \nu - \mu : p(\vec{u}) \rangle, \gamma')}$$

application

$$\frac{\mathcal{X} \models c \Rightarrow z = y}{\begin{array}{c}(c|\kappa_1, \langle \mu - closure(x, \mu' : A, z) \rangle, \kappa_2 | \gamma, \langle \nu - \nu : apply(y,t) \rangle, \gamma') \longrightarrow \\ (c|\kappa_1, \langle \mu - closure(x, \mu' : A, z) \rangle, \kappa_2 | \gamma, \langle \mu - \mu' : A[x \backslash t] \rangle, \gamma')\end{array}}$$

TAB. 6.1 – Relation de transition pour les buts modulaires avec méta-appels et fermetures.

la règle *application* rend possible le réveil de fermetures collectées au cours de la dérivation, comme cela est attendu dans l'exemple 6.2 pour la définition de `forall`. En pratique cette règle cherche une fermeture associée à la variable de fermeture z (formellement elle vérifie que $y = z$), substitue la variable d'abstraction par l'argument appliqué et exécute le résultat dans le contexte où la fermeture a été crée.

6.3.3 Protection du code

Intuitivement la protection du module appelé est une propriété qui énonce que seuls les prédicats publics d'un module μ peuvent être appelés depuis l'extérieur, les sous-buts produits par cet appel étant exécutés dans le contexte μ. La protection du module appelant est une propriété qui énonce qu'un but contenu dans une fermeture ne peut être exécuté que dans le contexte de création de ladite fermeture. Ces deux propriétés peuvent être formalisées de la façon suivante :

Définition 6.14 (Protection du module appelé)**.** La sémantique opérationnelle d'un programme modulaire satisfait la propriété de *protection du module appelé* si la réduction d'un atome qualifié $\mu : p(\vec{t})$ dans un contexte ν produit des atomes qualifiés et des fermetures dans le contexte μ seulement et si p

est public dans μ ou $\mu = \nu$.

Définition 6.15 (Protection du module appelant)**.** La sémantique opérationnelle d'un programme modulaire satisfait la propriété du *protection du module appelant* si l'application d'une fermeture créée dans un contexte ν produit seulement des atomes et des fermetures dans le contexte ν.

Théorème 6.16. *La sémantique opérationnelle de MPLC satisfait la protection du module appelé et la protection du module appelant.*

Preuve. Pour la propriété de protection du module appelé, on peut considérer la réduction d'un atome qualifié $\mu{:}p(\vec{t})$ dans un contexte ν. Dans ce cas, seules les règles *CSLD modulaire* et *méta-appel* peuvent s'appliquer, produisant ainsi un but dans le contexte μ'. Dans le premier cas, nous avons que $\mu' = \mu$ et que p est public dans μ ou $\mu = \nu$. Dans le second cas, on a trivialement $\mu = \nu = \nu'$.

Pour la propriété de protection du module appelant, on remarque tout d'abord que les règles qui définissent la transition $\longrightarrow$ ne changent pas le contexte d'une fermeture, qui reste ainsi dans le contexte dans lequel elle a été crée. On conclut en constatant que pour une application de fermeture créée dans un contexte ν, la règle *application* est la seule règle applicable et qu'elle produit un but dans le contexte ν. □

6.4 Conclusion

Nous avons illustré comment l'utilisation des fermetures pouvait élégamment résoudre le problème de protection du code que beaucoup de systèmes préfèrent simplement ignorer. Nous avons proposé un système de modules proche de celui de Ciao Prolog dans son implémentation. Ce système à été défini avec une sémantique opérationnelle qui a été utilisée pour prouver formellement une propriété de protection du code. Néanmoins aucune sémantique logique propre, incluant méta-appels et fermeture n'a encore été proposée. Nous montrerons dans le chapitre suivant comment les languages LCC apportent une solution logiquement satisfaisante à ce problème.

Ce système[2] à été implémenté au-dessus de GNU-Prolog RH [DH05], une extension de GNU-Prolog avec attributs et contraintes sur les nombres réels. Le système obtenu a été utilisé pour porter certainnes bibliothèques préexistantes. Par exemple l'implémentation des CHR de Schrijvers [SW04] qui a été portée sur le prototype, fournit une exemple intéressant d'utilisation intensives des modules.

[2]abandonné au profit du système présenté en fin de chapitre 9.

Chapitre 7

Un Système de Modules Internalisé dans les langages LCC

Sommaire

7.1 Introduction

Afin de permettre la protection des données et du code d'un agent LCC, ce chapitre introduit un système de modules particulièrement simple. Il s'agit en fait d'une couche de sucre syntaxique qui restreint l'ensemble des agents LCC définissables. Comme cela a été fait pour les déclarations et les fermetures, les modules sont internalisés aux agents et pourront être utilisés comme des objets de première classe.

Ce chapitre, dont une partie a été publiée dans [HFS07], se décompose comme suit. Dans la première section la syntaxe des agents LCC modulaires est introduite, leur sémantique opérationnelle étant définie par une interprétation en LCC sans modules. La première section introduit les conventions syntaxiques de MLCC, une version modulaire de LCC. La sous-section 7.2.4, définit la notion de contrainte décomposable qui servira tout au long du chapitre. L'ensemble des variables accessibles par un agent dans le store est ensuite défini Afin de définir une classe d'agents capables de préserver l'inaccessibilité de leurs variables vis-à-vis d'agents externes, un système de type élémentaire est introduit. La dernière section présente formellement une notion de protection du code.

7.2 Les Langages LCC Modulaires

Soient $\Sigma_{\mathcal{D}}$ et $\Sigma_{\mathcal{M}}$ deux alphabets de symboles de contrainte supposés disjoints, $\mathcal{D}$ et $\mathcal{M}$ les langages de contraintes construits à partir de ces alphabets respectifs. On supposera que $(\mathcal{D}, \Vdash_{\mathcal{D}})$ est un système de contraintes, contenant l'égalité $=$, construit comme l'encodage d'un système de contraintes classique tel que défini à la section 3.2.2 ; ce système étant supposé respecter la cohérence de l'égalité (cf. définition 3.7). On supposera que $(\mathcal{C}, \Vdash_{\mathcal{C}})$ est le système de contraintes obtenu en ajoutant à $(\mathcal{D}, \Vdash_{\mathcal{D}})$ les contraintes de synchronisation construites à partir de $\Sigma_{\mathcal{M}}$.

Dans la suite, on notera c, d, $e\ldots$ les contraintes, supposées sans bang, de $\mathcal{C}$ et m, n,... les contraintes de $\mathcal{M}$. Les contraintes de $\mathcal{D}$ (resp. de $\mathcal{M}$) seront appelées contraintes *classiques* (resp. contraintes *modulaires*).

7.2.1 Conventions syntaxiques

L'internalisation des déclarations et le mécanisme de fermeture fournis par les *ask* persistants de LCC peuvent être utilisés pour construire un système de modules complet pour les langages LCC. Dans cette approche, un module est référencé par un terme. Bien que, en théorie, rien n'empêche d'utiliser

des termes quelconques comme références, on se contentera, en pratique, de référencer un module par une variable libre ou une constante de Σ_F. Les variables représenteront des modules anonymes alors que les constantes de Σ_F représenteront des modules globals. Ainsi il sera possible d'appeler un module nommé par une constante même si celui-ci n'est pas défini dans le scope courant (cas typique d'une bibliothèque compilée séparément)[1].

La convention syntaxique $t\{A\}$ sera utilisée pour représenter l'agent A dans le module référencé par t. De façon similaire l'ajout d'une contrainte de synchronisation $m \in \mathcal{M}$ dans le module t sera dénoté par $t\!:\!m$, alors que l'ajout d'une contrainte classique $d \in \mathcal{D}$ ne sera pas localisé (il sera toujours global).

Définition 7.1 (Variable quantifiable)**.** Une variable *quantifiable* dans une contrainte de $\mathcal{C}$, est une variable qui apparaît au moins une fois dans une sous-contrainte modulaire de c. Formellement l'ensemble de variables *quantifiables* dans une contrainte de $\mathcal{C}$ sans bang est défini récursivement comme :

- $\mathcal{V}_{\mathcal{M}}(m(\vec{t})) = \mathcal{V}(\vec{t})$ si $p \in \Sigma_{\mathcal{M}}$
- $\mathcal{V}_{\mathcal{M}}(d(\vec{t})) = \emptyset$ si $d \in \Sigma_{\mathcal{D}}$
- $\mathcal{V}_{\mathcal{M}}(c_1 \otimes c_2) = \mathcal{V}_{\mathcal{M}}(c_1) \cup \mathcal{V}_{\mathcal{M}}(c_2)$,
- $\mathcal{V}_{\mathcal{M}}(\exists x.c) = \mathcal{V}_{\mathcal{M}}(()c) \setminus \{x\}$

où d est un symbole de $\Sigma_{\mathcal{D}}$, m est un symbole de $\Sigma_{\mathcal{M}}$ et c_1, c_2 et c sont des contraintes de $\mathcal{C}$.

Définition 7.2 (Conventions Syntaxiques)**.** La syntaxe des *agents* des langages LCC modulaires (notés MLCC) est définie par la grammaire suivante :

$$A ::= t\{A\} \mid x\!:\!m \mid m \mid d \mid A \parallel A \mid \exists t.A \mid \forall \vec{x}(c \rightarrow A) \mid \forall \vec{x}(c \Rightarrow A)$$

où les contraintes m, d et c sont des contraintes (sans bang) appartenant respectivement à $\mathcal{M}$, $\mathcal{D}$ et $\mathcal{C}$. De plus, seules les variables quantifiables sont supposées être quantifiées dans les gardes des *ask* (formellement $\vec{x} \subset \mathcal{V}_{\mathcal{M}}(c)$).

Exemple 7.3. Un module *list* peut être défini avec un implémentation interne anonyme pour le prédicat *reverse* :

$$\begin{array}{l} list\{\exists Impl.(\\ \quad \forall X, Y.reverse(X, Y) \Rightarrow Impl\!:\!reverse(X, [], Y) \parallel \\ \quad Impl\{ \\ \quad\quad \forall X.reverse([], X, X) \Rightarrow true \parallel \end{array}$$

[1] L'utilisation d'un terme composé est discuté dans la partie discussion (Section 7.6 de ce chapitre).

$$\forall X, Y, Z, T. reverse([X|Y], Z, T) \Rightarrow reverse(Y, [X|Z], T).$$
$$\}$$
$$)\}$$

Dans la suite, il sera démontré que dans une telle implémentation de *list*, seul le prédicat *reverse*/2 peut être appelé de l'extérieur du module, ainsi la version de *reverse* avec accumulateur, reste cachée pour le reste du programme.

7.2.2 Interprétation

Les agents MLCC($\mathcal{C}$) sont traduits en agents LCC sur un système de contraintes modifié, noté $\dot{\mathcal{C}}$, dans lequel un argument supplémentaire est ajouté à chaque contrainte de synchronisation. De même on notera $\dot{\mathcal{M}}$ le langage des contraintes de synchronisation de $\mathcal{M}$ auxquelles ont été ajoutées un argument.

Définition 7.4 (Interprétation dans LCC)**.** Pour tout terme t référençant un module, la traduction $()^t$ des agents MLCC($\mathcal{C}$) (resp. des contraintes de $\mathcal{C}$) vers les agents LCC($\dot{\mathcal{C}}$) (resp. les contraintes de $\dot{\mathcal{C}}$) est définie récursivement comme :

$$d(\vec{s})^t = d(\vec{s}) \qquad p(\vec{s})^t = \dot{p}(t, \vec{s}) \qquad (\exists y.c)^t = \exists y.c^t \qquad (c_1 \otimes c_2)^t = c_1^t \otimes c_2^t$$

$$(t\{A\})^s = A^t \qquad (t{:}m)^s = m^t \qquad (A \parallel B)^s = A^s \parallel B^s \qquad (\exists x.A)^s = \exists x.A^s$$

$$(\forall \vec{y}(c \rightarrow A))^s = \forall \vec{x}(c^s \rightarrow A^s) \qquad (\forall \vec{y}(c \Rightarrow A))^s = \forall \vec{x}(c^s \Rightarrow A^s)$$

où d et m sont des symboles respectifs de $\Sigma_{\mathcal{D}}$ et $\Sigma_{\mathcal{M}}$, c, c_1, c_2 sont des contraintes quelconques de $\mathcal{C}$ et, sans perte de généralité, $\mathcal{V}(\vec{y}, x) \cap \mathcal{V}(s) = \emptyset$.

La syntaxe de MLCC impose en pratique une restriction sur l'ensemble des agents LCC définissables. En employant la syntaxe de MLCC, il n'est pas possible de quantifier le premier argument d'une contrainte de $\dot{\mathcal{M}}$ quand elle apparaît dans la garde d'un *ask* (persistant ou non). Par exemple $\forall \vec{x}(m(\vec{x}) \Rightarrow A)$ est interprété, dans un module t comme $\forall \vec{x}(\dot{m}(t, \vec{x}) \Rightarrow \dot{A})$ avec $\mathcal{V}(t) \cap \vec{x} = \emptyset$ alors que $\forall y \vec{x}(\dot{m}(y, \vec{x}) \Rightarrow \dot{B})$ n'est la traduction d'aucun agent MLCC . Il sera démontré dans la suite que, grâce à cette restriction, il est possible de restreindre l'ensemble des variables accessibles par un *ask*. Il sera alors possible de garantir une forme de protection du code, comme nous l'avons fait pour la PLC.

Exemple 7.5. L'interprétation de l'agent MLCC($\mathcal{C}$) défini dans l'exemple 7.3 est l'agent LCC($\mathcal{C}$) suivant :

$$
\begin{array}{l}
\exists Impl.(\\
\quad \forall X, Y.reverse(list, X, Y) \Rightarrow reverse(Impl, X, [], Y) \parallel \\
\quad \forall X.reverse(Impl, [], X, X) \Rightarrow true \parallel \\
\quad \forall X, Y, Z, T.reverse(Impl, [X|Y], Z, T) \Rightarrow reverse(Y, [X|Z], T). \\
)
\end{array}
$$

Définition 7.6 (Agent modulaire)**.** Un agent LCC($\dot{\mathcal{C}}$) A est *modulaire* s'il est la traduction d'un agent MLCC($\mathcal{C}$), c.-à-d. s'il existe un agent MLCC($\mathcal{C}$) B et un terme t tel que $A = B^t$. Une configuration sur $LCC(\dot{\mathcal{C}})$ est dite *modulaire* si ses agents sont des agents modulaires. Un contexte de LCC($\dot{\mathcal{C}}$) est modulaire si le remplacement de ses trous par des agents modulaires quelconques donne un agent modulaire.

Remarque 7.7. Le réduit d'un agent modulaire est un agent modulaire.

Dans la suite, les contraintes de $\dot{\mathcal{C}}$ seront dénotées par $\dot{c}$, $\dot{d}$, $\dot{e}$, ... De même, par soucis de lisibilité les agents de $MLCC(\mathcal{C})$ seront notés, dans la suite, A, B, ... alors que les agents de $LCC(\dot{\mathcal{C}})$ seront notés $\dot{A}$, $\dot{B}$, ...

7.2.3 Exemples

Itérateurs

L'encodage des itérateurs en LCC (cf. exemple 4.1) peut être évidemment transcrit en MLCC. Dans le cas modulaire, une fermeture sera représentée par un module contenant, par convention, un *ask* de la forme $y\{\forall x(apply(x) \Rightarrow A)\}$. L'application de la fermeture se fait alors par le *tell* modulaire $y : apply(x)$ ou y est la référence de la fermeture et x l'argument appliqué.

Exemple 7.8 (Itérateurs de listes en MLCC). :

$$
\begin{array}{l}
list\{ \\
\quad \forall y(forall([], y) \Rightarrow \mathbf{1}) \parallel \\
\quad \forall x_H x_T y(forall([x_H|x_T], y) \Rightarrow (y : apply(x_H) \parallel forall(x_T, y)) \parallel \\
\\
\quad \forall x_H x_T y(exists([x_H|x_T], y) \Rightarrow y : apply(X_H)) \parallel \\
\quad \forall x_H x_T y(exists([x_H|x_T], y) \Rightarrow exists(x_T, y)) \\
\}
\end{array}
$$

On notera que l'interprétation (en utilisant la définition 7.2) de cet agent en LCC est exactement identique à l'agent de l'exemple 4.1 présenté dans la première partie de ce mémoire.

Le banquet des philosophes

Pour démontrer le gain expressif permis par les modules, reprenons l'exemple 3.24. Le mécanisme de module permet d'étendre le dîner des philosophes en un banquet où plusieurs tables de philosophes sont réunies, chaque table étant représentée par un module indépendant. L'agent MLCC ci-dessous crée n tables de p philosophes, quand il placé en composition parallèle avec l'agent $banquet : recTbale(0, n, p)$.

Exemple 7.9. (Banquet des philosophes).

$$
\begin{array}{l}
banquet\{ \\
\quad \forall x_i x_n x_p . recTable(x_i, x_n, x_p) \Rightarrow \\
\qquad \exists y_{table} . y_{table}\{ \\
\qquad\quad \forall x_j . recPhilo(x_j) \Rightarrow (\\
\qquad\qquad y_{table} : fork(x_j) \parallel \\
\qquad\qquad fork(x_j) \otimes fork(x_j + 1 \bmod x_p) \Rightarrow \\
\qquad\qquad\quad y_{table} : eat(x_j) \parallel \\
\qquad\qquad eat(x_j) \Rightarrow \\
\qquad\qquad\quad y_{table} : (fork(x_j) \otimes fork(x_j + 1 \bmod x_p)) \parallel \\
\qquad\qquad x_j \neq x_p \multimap y_{table} : recPhilo(x_j + 1)\ ) \parallel \\
\qquad\quad y_{table} : recPhilo(0) \\
\qquad \} \parallel \\
\qquad x_i \neq x_n \multimap banquet : recTable(x_i + 1, x_n, x_p) \\
\}
\end{array}
$$

Puisque les langages MLCC sont définis par une simple couche de sucre syntaxique au-dessus des langages LCC, ils possèdent bien évidemment les mêmes propriétés de sémantique logique. En conséquence, la sémantique des phases de la logique linéaire peut par exemple être utilisée de façon similaire à [FRS01] pour prouver de propriétés de sûreté, telles que, par exemple, aucun philosophe ne peut utiliser une fourchette disposée sur une autre table.

7.2.4 Contraintes décomposables

Certains résultats, de ce chapitre, seront conditionnés par la *décomposabilité* des stores issus des dérivations des configurations modulaires considérées : il doit être possible de séparer dans un store $\dot{c} \in \dot{\mathcal{C}}$, les contraintes modulaires des contraintes classiques et cela sans affaiblir ledit store. Cette section, prouve qu'à partir d'un store initial vide (c.-à-d. $\mathbf{1}$) les agents modulaires ne produisent que des stores possédant cette propriété.

Définition 7.10 (Contrainte décomposable)**.** Une contrainte de la forme $d \otimes m_1^{s_1} \otimes \cdots \otimes m_k^{s_k}$ (avec $k \geq 0$) est une contrainte sous *forme séparée* si les m_i sont des contraintes atomiques de $\mathcal{M}$ et d une contrainte de $\mathcal{D}$. Une contrainte sous *forme décomposée* est une contrainte de $\dot{\mathcal{C}}$ de la forme $\exists Y.(\dot{c})$ où $\dot{c}$ est sous forme séparée. Une contrainte de $\mathcal{C}$ est *décomposable* (resp. *séparable*) si elle est équivalente à une contrainte sous forme décomposée (resp. séparée).

Notons que le store $\mathbf{1}$ est trivialement décomposable.

Lemme 7.11 (Stabilité de la décomposabilité par implication)**.** *Soient* Γ *un multiensemble cohérent (c.-à-d.* $\Gamma \not\vdash_{\dot{\mathcal{C}}} \mathbf{0}$*) de contraintes sous* forme décomposée *et* $\dot{c}$ *une contrainte de* $\dot{\mathcal{C}}$*.*

(i) Si $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}$ *alors* $\dot{c}$ *est décomposable.*

(ii) Si $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c} \otimes \dot{T}$ *alors* $\dot{c}$ *est décomposable.*

(iii) Si $\Gamma \vdash_{\dot{\mathcal{C}}} \exists Y.(\dot{c})$ *alors* $\dot{c}$ *est décomposable.*

Preuve. Le cas (i) se prouve par induction la preuve π de $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}$. Grâce à la proposition 3.35, on sait que cette induction à un sens.

- π est un axiome. Il y deux cas suivant la nature de l'axiome :
  - axiome logique : sans perte de généralité, $\dot{c}$ est atomique et donc trivialement sous forme décomposée.
  - axiome non-logique. Il suffit de constater que $\dot{c}$ est soit une contrainte classique si l'axiome provient de l'encodage du système de contraintes classique ou une contrainte modulaire s'il s'agit d'un axiome pour la substitution dans les contraintes de synchronisation vis-à-vis de l'égalité. $\dot{c}$ est donc trivialement sous forme décomposée.
  - π finit par une coupure : la coupure est, sans perte de généralité, d'une des deux formes suivantes :

$$\frac{\Gamma \vdash_{\dot{\mathcal{C}}} \dot{d} \quad \overline{\dot{d} \vdash_{\dot{\mathcal{C}}} \dot{c}}}{\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}} \qquad \text{ou} \qquad \frac{\overline{\dot{e} \vdash_{\dot{\mathcal{C}}} \dot{d}} \quad \dot{d}, \Delta \vdash_{\dot{\mathcal{C}}} \dot{c}}{\dot{e}, \Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}}$$

    Dans les deux cas, un raisonnement identique aux cas précédents nous permet de conclure que, dans le sous-cas de gauche (resp. dans le sous-cas de droite), $\dot{c}$ (resp. $\dot{d}$) est sous forme décomposée. Il suffit alors, d'appliquer l' hypothèse d'induction pour le sous-cas de droite.

– π finit par une *promotion* :

$$\frac{!\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}}{!\Gamma \vdash_{\dot{\mathcal{C}}} !\dot{c}}$$

Notons que dans ce cas $!\Gamma$ ne contient que des contraintes classiques, sinon elles ne seraient pas sous forme décomposée. $!c$ est alors nécessairement une contrainte classique et donc décomposable.

– π finit par une introduction à droite du $\otimes$:

$$\frac{\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c} \quad \Delta \vdash_{\dot{\mathcal{C}}} \dot{d}}{\Gamma, \Delta \vdash_{\dot{\mathcal{C}}} \dot{c} \otimes \dot{d}}$$

Par hypothèse d'induction, $\dot{c}$ et $\dot{d}$ sont décomposables, ainsi il existe $\dot{c}'$ et $\dot{d}'$ deux contraintes sous forme séparée et X et Y deux ensembles de variables tels que $\dot{c} \dashv\vdash_{\dot{\mathcal{C}}} \exists X.\dot{c}'$ et $\dot{d} \dashv\vdash_{\dot{\mathcal{C}}} \exists Y.\dot{d}'$. Sans perte de généralité $X \cap Y = \emptyset$ et donc $\dot{c} \otimes \dot{d} \dashv\vdash_{\dot{\mathcal{C}}} \exists XY.\dot{c}' \otimes \dot{d}'$

– les autres cas sont des implications directes de l'hypothèse d'induction.

Pour (ii) on procède par induction sur la preuve de $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c} \otimes \top$. Le seul cas réellement intéressant est l'introduction à droite du $\otimes$ qui sépare $\dot{c}$ et $\top$:

$$\frac{\Gamma, \vdash_{\dot{\mathcal{C}}} \dot{c} \quad \Delta \vdash_{\dot{\mathcal{C}}} \top}{\Gamma, \Delta \vdash_{\dot{\mathcal{C}}} \dot{c} \otimes \top}$$

Pour ce cas il suffit d'appliquer le résultat précédent.

Pour (iii) l'induction est analogue à celle du sous-cas (i). Il suffit de constater que si $\exists x.c$ appartient à $\mathcal{D}$ (resp. $\dot{\mathcal{M}}$) alors c appartient à $\mathcal{D}$ (resp. $\dot{\mathcal{M}}$). □

Proposition 7.12 (Stabilité de la décomposabilité par réduction)**.** *Soient* $\langle X; \dot{c}; \Gamma\rangle$ *et* $\langle Y; \dot{d}; \Delta\rangle$ *deux configurations modulaires telles que* $\langle X; \dot{c}; \Gamma\rangle \longrightarrow \langle Y; \dot{d}; \Delta\rangle$ *soit une dérivation cohérente.*

Si $\dot{c}$ *est décomposable alors* $\dot{d}$ *est décomposable.*

Preuve. On procède par induction sur la longueur de $\langle X; \dot{c}; \Gamma\rangle \xrightarrow{*} \langle Y; \dot{d}; \Delta\rangle$

– Le cas de base est immédiat.

– La dernière règle est un *tell*. Il y a deux cas, suivant la nature de l'agent réduit, qui est soit une pose de contrainte classique soit une pose de contraintes modulaire.Dans les deux cas la contrainte ajoutée au store est trivialement décomposable, il suffit donc d'utiliser le lemme précédent.

– La dernière règle est une *localisation*. Immédiat car la règle ne touche pas au store.

– La dernière règle est un *ask* (persistant ou non) :

$$\langle X;\, \dot{c};\, \Gamma\rangle \xrightarrow{*} \langle X \cup Y;\, \dot{e};\, \Delta\rangle$$

avec $\dot{c}' \vdash_{\dot{\mathcal{C}}} \exists Y(\dot{d} \otimes \dot{e})$ où d est une instance de la garde du *ask* réduit. Par hypothèse d'induction, on sait que $\dot{c}$ est décomposable. Il suffit alors utiliser le lemme précédent. □

Ce résultat garantit que les réductions de configurations modulaires préservent la décomposabilité du store de contraintes. Il est alors possible de supposer, sans perte de généralité, que toutes les configurations considérées dans la suite ont des stores décomposables.

7.3 Accessibilité des Variables

Cette section s'intéresse aux variables non fraîches, qu'un agent peut récupérer au cours d'une réduction. Dans le cas de LCC, tel que présenté au chapitre 3, nous avons vu que l'opérateur $\forall$ du *ask* permettait à un agent de récupérer les variables d'un agent tiers. Cette section montre que les restrictions syntaxiques imposées par les conventions syntaxiques de MLCC restreignent l'ensemble des variables accessibles par un agent.

Pour cela, une notion ad-hoc d'accessibilité de variables est tout d'abord introduite.

7.3.1 Définition

Définition 7.13 (Accessibilité)**.**

– L'ensemble des variables *accessibles par unification* dans une contrainte $\dot{c}$ de $\dot{\mathcal{C}}$ à partir d'un ensemble de variables X est l'ensemble :

$$\mathcal{A}^u_{\dot{c}}(X) = X \cup \left\{ x \in \mathcal{V}(\dot{c}) \;\middle|\; \begin{array}{c} d \in \mathcal{D} \quad y \in \mathcal{V}(d) \quad \mathcal{V}(\dot{c}) \cap \mathcal{V}(d) \subset X \\ \dot{c} \otimes d \vdash_{\mathcal{C}} x = y \otimes \top \qquad \Gamma, d \not\vdash_{\mathcal{C}_c} \mathbf{0} \end{array} \right\}$$

– L'ensemble des variables *accessibles par substitution* dans une contrainte $\dot{c}$ sous forme séparée à partir d'un ensemble de variables X est l'ensemble :

$$\mathcal{A}^s_{\dot{c}}(X) = X \cup \left\{ x \in \mathcal{V}(\vec{t}) \,\middle|\, \dot{m} \in \Sigma_{\mathcal{M}}, \mathcal{V}(s) \subset X, \dot{c} \vdash_{\dot{\mathcal{C}}} \dot{m}(s, \vec{t}) \otimes \top, \dot{c} \not\vdash_{\dot{\mathcal{C}}} \mathbf{0} \right\}$$

– L'ensemble des variables *directement accessibles* dans une contrainte sous forme séparée $\dot{c} = d \otimes m_1^{s_1} \otimes \cdots \otimes m_k^{s_k}$ à partir d'un ensemble de variables X est défini comme l'ensemble :

$$\mathcal{A}^1_{\dot{c}}(X) = \mathcal{A}^u_{\dot{c}}(X) \cup \mathcal{A}^s_{\dot{c}}(X)$$

– L'ensemble des variables *accessibles* dans une contrainte sous forme séparée $\dot{c}$ de $\dot{\mathcal{C}}$, noté $\mathcal{A}_{\dot{c}}(X)$ est le plus petit point fixe de $\mathcal{A}^1_{\dot{c}}$ contenant X.

– L'ensemble des variables *accessibles* dans une contrainte décomposable $\dot{c}$ de $\dot{\mathcal{C}}$ est l'ensemble $\mathcal{A}_{\dot{c}}(X) = \mathcal{A}_{\dot{d}}(X) \setminus Y$ où $\dot{d} \in \dot{\mathcal{C}}$ et Y sont respectivement une contrainte sous forme séparée et un ensemble de variables tel que $\dot{c} \vdash_{\dot{\mathcal{C}}} \exists Y.\dot{d}$, avec sans perte de généralité, $X \cap Y = \emptyset$.

La proposition suivante nous assure que le calcul de point fixe utilisé dans la définition précédente a un sens.

Proposition 7.14. *Pour toute contrainte sous forme décomposée $\dot{c}$, $\mathcal{A}^1_{\dot{c}}$ est extensive, croissante et bornée.*

Preuve. Par définition $\mathcal{A}^u_{\dot{c}}$ et $\mathcal{A}^s_{\dot{c}}$ sont extensives, donc $\mathcal{A}^1_{\dot{c}}$ l'est aussi. Pour la croissance, il suffit de constater que pour toute paire d'ensembles de variables X et Y et toute variable x si $x \in \mathcal{A}^u_{\dot{c}}(X)$ (resp. $x \in \mathcal{A}^s_{\dot{c}}(X)$) alors $x \in \mathcal{A}^u_{\dot{c}}(X \cup Y)$ (resp. $x \in \mathcal{A}^s_{\dot{c}}(X \cup Y)$). Pour conclure il suffit de remarquer que $\mathcal{V}$ est bien évidemment un majorant de $\mathcal{A}^1_{\dot{c}}$. □

Exemple 7.15. Soit $c = (x_1 < x_2 \otimes z_1 < z < z_2)$ une contrainte classique sur les entiers naturels.

– z est accessible à partir de $\{z_1, z_2\}$, en effet $c \otimes d \vdash_{\mathcal{C}} z = y \otimes \top$ avec $d = (z_1 < y < z_2 \otimes z_1 = z_2 + 2)$.

– x_2 et x_1 ne sont pas accessibles à partir de $\{z, z_1, z_2\}$

Remarque 7.16. Quel que soit $\dot{c}$ une contrainte décomposable, on a :

$$\mathcal{A}_{\dot{c}}(X) \subset (X \cup \mathcal{V}(\dot{c})).$$

Remarque 7.17. Soient X' et Y' deux ensembles de variables. Si $Y' \notin \mathcal{V}(\dot{c})$ alors $\mathcal{A}_{\dot{c}}(X') \cup Y' = \mathcal{A}_{\dot{c}}(X' \cup Y')$.

7.3.2 Monotonie de l'accessibilité

Cette sous-section se concentre sur la démonstration de la monotonie de la définition d'accessibilité. Il sera ainsi prouvé qu'une variable non accessible au début d'une dérivation ne sera jamais accessible.

Lemme 7.18. *Soient $\dot{c} \in \dot{\mathcal{C}}$ et $\dot{d} \in \dot{\mathcal{C}}$ deux contraintes cohérentes et X un ensemble arbitraire de variables.*

(i) Si $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d} \otimes \top$ et $\dot{c}$ et $\dot{d}$ sont sous forme séparée alors $\mathcal{A}_{\dot{c}}(X) \supset \mathcal{A}_{\dot{d}}(X)$;

(ii) Si $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}$ et $\dot{c}$ est décomposable alors $\mathcal{A}_{\dot{c}}(X) \supset \mathcal{A}_{\dot{d}}(X)$;

(iii) Si $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d} \otimes \top$ et $\dot{c}$ est décomposable alors $\mathcal{A}_{\dot{c}}(X) \supset \mathcal{A}_{\dot{d}}(X)$.

Preuve. Comme $\dot{c}$ est décomposable et à l'aide du lemme 7.11, on peut supposer sans perte de généralité, que $\dot{c}$ est sous forme décomposée. On procède alors par cas :

(i) Pour prouver que $\mathcal{A}^u_{\dot{c}}(X) \supset \mathcal{A}^u_{\dot{d}}(X)$, il suffit de constater que si $\dot{d} \otimes e \vdash_{\mathcal{C}} x = y \otimes \top$ alors $c \otimes e\rho \vdash_{\mathcal{C}} x = y\rho$ pour tout renommage ρ vérifiant :
   - si $z \in \mathcal{V}(\dot{d})$ alors $z\rho = z$
   - si $z \in \mathcal{V}(\dot{e}) \setminus \mathcal{V}(\dot{d})$ alors $z\rho \notin \mathcal{V}(\dot{c}, \dot{d})$.

   La preuve de $\mathcal{A}^s_{\dot{c}}(X) \supset \mathcal{A}^s_{\dot{d}}(X)$ est une conséquence immédiate des hypothèses. Compte tenu de ces deux résultats, on a $\mathcal{A}^1_{\dot{c}}(X) \supset \mathcal{A}^1_{\dot{d}}(X)$, on conclut le cas grâce à l'extensitivité de $\mathcal{A}^1_{\dot{d}}$.

(ii) On procède par induction sur la preuve π de $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}$:
   - π finit par un séquent de la forme $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}$ où $\dot{c}$ et $\dot{d}$ sont sous forme séparée : il suffit d'utiliser le cas précédent.
   - π finit par une introduction du $\exists$:

$$\frac{\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}}{\exists x.\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}} x \notin \mathcal{V}(\dot{d}) \qquad \text{ou} \qquad \frac{\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}[x \backslash t]}{\dot{c} \vdash_{\dot{\mathcal{C}}} \exists x.\dot{d}}$$

     Dans les deux cas il suffit d'utiliser l'hypothèse d'induction et la remarque 7.16.
   - π finit par une introduction à gauche du $\otimes$ de la forme :

$$\frac{\dot{c}_1, \dot{c}_2 \vdash_{\dot{\mathcal{C}}} \dot{d}}{\dot{c}_1 \otimes \dot{c}_2 \vdash_{\dot{\mathcal{C}}} \dot{d}} \qquad \text{où } \dot{d} \text{ n'est pas sous forme séparée}$$

     Puisque $\dot{d}$ n'est pas sous forme séparée, elle est nécessairement de la forme $\exists y.\dot{e}$. Il suffit de constater que la règle qui élimine le $\exists$ à droite peut être permutée avec la dernière règle de π, on se retrouve alors dans le cas précédent.

– π finit par une introduction à droite du $\otimes$:

$$\frac{\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}_1 \quad \vdash_{\dot{\mathcal{C}}} \dot{d}_2}{\dot{c} \vdash_{\mathcal{C}} \dot{d}_1 \otimes \dot{d}_2}$$

Dans ce cas, on constate que d_2 est une contrainte classique et que $\mathcal{A}^u_{\dot{d}_2}(X) = X$; sinon la cohérence de l'égalité ne serait pas assurée. On conclut en utilisant l'hypothèse d'induction.

(iii) Il y a essentiellement deux cas :

– $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d}$: Dans ce cas, il suffit d'utiliser *(ii)*.

– $\dot{c} \nvdash_{\dot{\mathcal{C}}} \dot{d}$: Dans ce cas, la preuve de $\dot{c} \vdash_{\dot{\mathcal{C}}} \dot{d} \otimes \top$ commence (de bas en haut) nécessairement par des éliminations a gauche de tous les $\exists$, puis des éliminations à gauche de $\otimes$ et enfin par l'élimination à droite du $\otimes$ qui sépare le membre gauche en deux multiensembles de contraintes $\dot{e}_1, \dots \dot{e}_i$ et $\dot{e}_{i+j}, \dots \dot{e}_n$ tels que $\dot{e}_1, \dots, \dot{e}_i \vdash_{\dot{\mathcal{C}}} \dot{d}$. Pour conclure il suffit de constater que dans ce cas $\dot{e}_1 \otimes \dots \otimes \dot{e}_i \vdash_{\dot{\mathcal{C}}} \dot{d}$ et que $\dot{e}_1 \otimes \dots \otimes \dot{e}_i$ est clairement décomposable. □

Lemme 7.19. *Pour toute contrainte décomposable $d \in \mathcal{C}$ et tout terme s on a $\mathcal{V}_m(d) \subset \mathcal{A}_{d^s}(\mathcal{V}(s))$.*

Preuve. On constate que si d est décomposable alors il existe une contrainte e telle que $d^s \dashv\vdash_{\dot{\mathcal{C}}} e^s$ et e^s est sous forme décomposée. On conclut par une simple induction sur la taille de e. □

Lemme 7.20. *Soient $\dot{c}$ une contrainte décomposable de $\dot{\mathcal{C}}$, d et $\dot{e}$ deux contraintes de $\mathcal{C}$ et $\dot{\mathcal{C}}$ et X et Y deux ensembles de variables tels que $Y \cap (\mathcal{V}(\dot{c}, s) \cup X) = \emptyset$. Si $\dot{c} \vdash_{\dot{\mathcal{C}}} \exists Y.(d^s \otimes \dot{e})$ alors :*

(i) $\mathcal{V}_m(d) \subset \mathcal{A}_{\dot{c}}(\mathcal{V}(s)) \cup Y$

(ii) Si $X \supset \mathcal{V}(s)$ *alors* $\mathcal{A}_{\dot{e}}(X \cup \mathcal{V}_m(d)) \subset \mathcal{A}_{\dot{c}}(X) \cup Y$

Preuve. On note tout d'abord que le lemme 7.18 impose que d^s et $\dot{e}$ soient décomposables. *(i)* est alors un corollaire des deux lemmes précédents.

Pour *(ii)*, l'application du lemme 7.18 permet d'inférer $\mathcal{A}_{\dot{e}}(X \cup \mathcal{V}_m(d)) \subset \mathcal{A}_{\dot{c}}(X \cup \mathcal{V}_m(d)) \cup Y$. Grâce à la remarque 7.17 on déduit alors que $\mathcal{A}_{\dot{e}}(X \cup \mathcal{V}_m(d)) \subset \mathcal{A}_{\dot{c}}(X \cup \mathcal{V}_m(d) \cup Y)$. L'utilisation de *(i)* et le fait que $\mathcal{A}_{\dot{e}}$ soit calculé par un point fixe permettent d'inférer que $\mathcal{A}_{\dot{e}}(X \cup \mathcal{V}_m(d)) \subset \mathcal{A}_{\dot{c}}(X \cup Y \cup \mathcal{V}_m(d))$. On conclut le cas *(ii)* en réutilisant 7.17, pour refaire sortir Y. □

Théorème 7.21 (Monotonie de l'accessibilité)**.** *Soit $\langle X; \dot{c}; \Gamma, \Delta\rangle \xrightarrow{*} \langle X'; \dot{c}'; \Gamma', \Delta\rangle$ une dérivation modulaire cohérente telle que Γ' soit le réduit de Γ et $\dot{c}$ une contrainte décomposable.*

$$Si \;\; x \in \mathcal{V}(X, \dot{c}, \Delta) \text{ et } x \notin \mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma)) \;\; alors \;\; x \notin \mathcal{A}_{\dot{d}}(\mathcal{V}(\Gamma'))$$

Preuve. On procède par induction sur la longueur de la dérivation.

- La dernière règle est un *tell* d'une contrainte modulaire :

$$\langle X;\, \dot{c};\, m^t, \Gamma, \Delta\rangle \longrightarrow \langle Y;\, \dot{c} \otimes m^t;\, \Gamma, \Delta\rangle$$

  Il suffit de constater que l'on a $\mathcal{A}_{\dot{c}}(\mathcal{V}(m^t, \Gamma)) = \mathcal{A}_{\dot{c}\otimes m^t}(\mathcal{V}(m^t, \Gamma))$ et $\mathcal{A}_{\dot{c}\otimes m^t}(\mathcal{V}(m^t, \Gamma)) \supset \mathcal{A}_{\dot{c}\otimes m^t}(\mathcal{V}(m^t))$.
- La dernière règle est un *tell* d'une contrainte classique :

$$\langle X;\, \dot{c};\, d, \Gamma', \Delta\rangle \longrightarrow \langle Y;\, \dot{c} \otimes d;\, \Gamma', \Delta\rangle$$

  On constate que pour toute contrainte classique $d \in \mathcal{D}$ on a $\mathcal{A}^u_{\dot{c}}(\mathcal{V}(d, \Gamma)) = \mathcal{A}^u_{\dot{c}\otimes d}(\mathcal{V}(d, \Gamma))$. Il suffit alors de noter que $\mathcal{A}^u_{\dot{c}\otimes d}(\mathcal{V}(d, \Gamma)) \supset \mathcal{A}^u_{\dot{c}\otimes d}(fv(\Gamma))$.
- Le cas de la *localisation* est immédiat, car la variable introduite n'est pas libres dans X, $\dot{c}$ et Δ.
- La dernière règle est un *ask* (persistant ou non) : Il suffit d'utiliser le lemme précédent. □

Ce théorème garantit que dans une configuration $\langle X;\, \dot{c};\, \Gamma, \Delta\rangle$, si Δ n'est pas réduit alors les variables libres de Δ qui ne sont pas accessibles par Γ dans $\dot{c}$ restent inaccessibles pour tous les réduits de Γ.

On peut alors se convaincre que si les agents de Δ ont été programmés avec suffisamment de soin, ils peuvent préserver l'inaccessibilité de certaines de leurs variables libres, vis-à-vis d'agents extérieurs.

7.3.3 Une classe naïve d'agents sûrs

Proposition 7.22. *Soit la dérivation* $\langle\{x\};\, \mathbf{1};\, \Gamma, \Delta\rangle \xrightarrow{*} \langle X;\, \dot{c};\, \Gamma', \Delta'\rangle$ *telle que* Γ' *et* Δ' *dérivent respectivement de* Γ *et* Δ *et que* x *n'apparaît que dans des tell modulaires de* Δ *de la forme* m^x *avec* $x \notin \mathcal{V}(m)$.

$$\textit{Si}\ \ x \notin \mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma))\ \ \textit{alors}\ \ x \notin \mathcal{A}_{\dot{d}}(\mathcal{V}(\Gamma'))$$

Preuve. On remarque tout d'abord que si x apparaît dans $\dot{c}$, il apparaît nécessairement comme premier argument d'une contrainte modulaire. Pour prouver le résultat on procède par induction sur la longueur de la dérivation :

- Le cas des réductions de Γ (ou d'un de ses réduits) est immédiat grâce à le théorème 7.21.
- Pour le cas de la réduction d'un *tell* dans A ou un de ses réduits, il suffit de constater que si y apparaît, dans $\dot{c}$, uniquement comme premier argument d'une contrainte modulaire, alors si $y \notin Y$ alors $y \notin \mathcal{A}_{\dot{c}}(Y)$.

- Le cas de la réduction d'une *localisation* dans A ou un de ses réduits est immédiat, car elle ne modifie pas le store.
- Le cas de la réduction d'un *ask* est une conséquence du lemme 7.18. □

Les agent décrits par cette proposition sont donc capables de conserver certaines de leurs variables inaccessibles vis-à-vis d'agents extérieurs. La classe ainsi définie n'est cependant pas satisfaisante, en effet les modules ne peuvent plus être manipulés comme des objets de première classe, puisqu'ils ne peuvent être utilisés comme argument. Un système de types simple, décrivant une classe de programmes plus générale sera présenté plus tard dans ce chapitre.

7.4 Protection du Code

Définition 7.23. Soit $\langle X;\, \dot{c};\, \Delta, \dot{A}, \dot{m}\rangle$ une configuration modulaire. La réduction de $\dot{B}$ *est indépendante* de la réduction de la contrainte de synchronisation $\dot{m}$ si pour toute dérivation qui réduit $\dot{m}$ puis $\dot{B}$, c.-à-d. de la forme :

$$\langle Y;\, \dot{c};\, \Delta, \dot{B}, \dot{m}\rangle \longrightarrow \langle Y;\, \dot{c} \otimes \dot{m};\, \Delta, \dot{B}\rangle \longrightarrow \langle Y';\, \dot{d};\, \Delta, \dot{B}'\rangle$$

il existe une dérivation qui réduit $\dot{B}$ puis $\dot{m}$ de la forme :

$$\langle Y;\, \dot{c};\, \Delta, \dot{B}, \dot{m}\rangle \longrightarrow \langle Y';\, \dot{e};\, \Delta, \dot{B}, \dot{m}\rangle \longrightarrow \langle Y';\, \dot{e} \otimes \dot{m};\, \Delta, \dot{B}'\rangle$$

avec $\dot{e} \otimes \dot{m} \vdash_{\dot{\mathcal{C}}} \dot{d}$.

Définition 7.24 (Protection du code)**.** Un agent modulaire $\dot{A}$ est *protégé* dans un contexte modulaire C si pour toute dérivation cohérente :

$$\langle \emptyset;\, \mathbf{1};\, C[\dot{A}], \Gamma\rangle \xrightarrow{*} \langle X;\, \dot{c};\, \Delta, \dot{B}_A, \dot{m}\rangle$$

où $\dot{B}_A$ dérive de $\dot{A}$ et $\dot{m}$ dérive de Γ, la réduction de $\dot{B}_A$ dans $\langle X;\, \dot{c};\, \Delta, \dot{B}_A, \dot{m}\rangle$ est indépendante de celle de $\dot{m}$.

Lemme 7.25. *Soit* $\langle X;\, \dot{c};\, \forall\vec{z}(d^x \rightsquigarrow \dot{A}), \Gamma, \Delta\rangle$ *une configuration modulaire où* $\forall\vec{z}(d^x \rightsquigarrow \dot{A})$ *est un ask persistant ou non. Si* $x \notin \mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma))$ *alors la réduction de* $\forall\vec{z}(d^x \rightsquigarrow \dot{A})$ *n'est imposée par aucune contrainte de synchronisation active dans* Γ.

Preuve. Le résultat se démontre par l'absurde : supposons que la réduction de $\forall\vec{z}(d^x \to \dot{A})$ (ou $\forall\vec{z}(d^x \Rightarrow \dot{A})$) soit imposée par une certaine contrainte de synchronisation active dans Γ. La contrainte $\dot{m}$ est donc consommée par le ask, c.-à-d. que $\dot{m}$ est de la forme $\dot{n}(y, \vec{t})$ avec $\dot{c} \vdash_{\mathcal{C}} (x = y) \otimes \top$. x est donc accessible à partir des variables libres de $\dot{m}$, ce qui contredit les hypothèses. □

Proposition 7.26 (Protection du code naïve)**.** *Soient A et B deux agents MLCC(C) et y une variable. Si A n'a pas de module interne et si la variable y n'est utilisé dans A et B que dans des* tell *modulaires de la forme $y : m$ avec $y \notin \mathcal{V}(m)$ alors A^t est protégé dans $(\exists y.(y\{A\} \parallel B))^t$ pour tout terme t.*

Preuve. On remarque tout d'abord que $(\exists y.(y\{A\} \parallel B))^t = \exists y.(A^y \parallel B^t)$ avec, sans perte de généralité, $y \notin \mathcal{V}(t)$. Soit $\langle \emptyset; \mathbf{1}; \exists y.(A^y \parallel B^t), \Gamma \rangle$ une configuration où Γ est un multiensemble arbitraire d'agents modulaires. Grâce à l'indépendance partielle vis-à-vis les stratégies de sélection des agents (théorème 3.31), il est possible de considérer sans perte de généralité, la configuration $\langle \{y\}; \mathbf{1}; A^y, B^t, \Gamma \rangle$ avec $y \notin fv(\Gamma)$. Soit la dérivation $\langle \{y\}; \mathbf{1}; A^y, B^t, \Gamma \rangle \xrightarrow{*} \langle Y; \dot{c}; \Delta, \Gamma' \rangle$ où Δ et Γ' dérivent respectivement de (A^y, B^t) et Γ. D'après la proposition 7.22, $y \notin \mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma))$. Il suffit alors d'utiliser le lemme précédent pour conclure. □

7.5 Typage d'Agents Sûrs

Cette section définit un typage d'agents sûrs, c.-à-d. de programmes capables de préserver l'inaccessibilité de certaines de leurs variables libres vis-à-vis d'agents externes. Ce typage repose sur la division de l'espace des variables d'une configuration en deux espaces préservés disjoints. Deux espaces de variables vont ainsi être utilisés, l'espace de variables *privées* que l'on souhaite préserver inaccessibles, et son dual l'espace de variables *publiques*.

7.5.1 Définition du système de types

Le typage suivant impose que les variables libres d'une contrainte classique postée dans le store soient toutes *privées* ou toutes *publiques*. Pour le cas des contraintes modulaires, qui représentent moralement des appels de prédicats, on impose le type de chacun de ses arguments par un environnement de typage. Ainsi on suppose que chaque symbole de contrainte modulaire m/k est fourni avec une signature du type $m : \tau_1 \times \cdots \times \tau_k$ où τ_i représente le statut du prédicat, privé noté **priv** ou public noté **pub**.

Définition 7.27 (Système de types)**.** Un *environnement* de typage est une fonction partielle qui associe aux symboles de $\Sigma_{\mathcal{M}}$ une signature notée $\{m_1 : \tau_1, \ldots m_n : \tau_n\}$, où chaque τ_i est une constante appartenant à $\{\mathbf{pub}, \mathbf{priv}\}$. Formellement le type d'un multiensemble d'agents est un ensemble de variables. On notera $\Sigma \triangleright \Delta : X$ le fait que Δ est de type X dans l'*environnement* Σ. Les règles du système de types pour les agents modulaires LCC est définit

par les règles suivantes :

$$\mathcal{D}\text{-}priv\ \frac{\mathcal{V}(d) \subset X \quad d \in \mathcal{D}}{\Sigma \triangleright d : X} \qquad \mathcal{D}\text{-}pub\ \frac{\mathcal{V}(d) \cap X = \emptyset \quad d \in \mathcal{D}}{\Sigma \triangleright d : X}$$

$$\mathcal{M}\text{-}priv\ \frac{\mathcal{V}(s) \subset X \quad m : \tau_1 \times \cdots \times \tau_n \in \Sigma \quad \left(\begin{array}{ll} \mathcal{V}(t_i) \subset X & \text{si } \tau_i = \text{priv} \\ \mathcal{V}(t_i) \cap X = \emptyset & \text{sinon} \end{array} \right)_{1 \leq i \leq n}}{\Sigma \triangleright \dot{m}(s, \vec{t_1}, \ldots, t_n) : X}$$

$$\mathcal{M}\text{-}pub\ \frac{\mathcal{V}(s, \vec{t}) \cap X = \emptyset}{\Sigma \triangleright \dot{m}(s, \ldots t) : X} \qquad \otimes\ \frac{\Sigma \triangleright \dot{A}, \dot{B} : X}{\Sigma \triangleright \dot{A} \otimes \dot{B} : X}$$

$$\exists\text{-}priv\ \frac{\Sigma \triangleright \dot{A} : X \cup \{x\}}{\Sigma \triangleright \exists x.\dot{A} : X} \qquad \exists\text{-}pub\ \frac{\Sigma \triangleright \dot{A} : X \quad x \notin X}{\Sigma \triangleright \exists x.\dot{A} : X}$$

$$\forall\text{-}priv\ \frac{\Sigma \triangleright \dot{A} : X \cup \{x\}}{\Sigma \triangleright \forall x.\dot{A} : X} \qquad \forall\text{-}pub\ \frac{\Sigma \triangleright \dot{A} : X \quad x \notin X}{\Sigma \triangleright \forall x.\dot{A} : X}$$

$$\| \frac{\Sigma \triangleright \dot{A}, \dot{B} : X}{\Sigma \triangleright \dot{A} \| \dot{B} : X} \qquad \rightarrow \frac{\Sigma \triangleright \dot{c}, \dot{A} : X}{\Sigma \triangleright \dot{c} \rightarrow \dot{A} : X} \qquad \Rightarrow \frac{\Sigma \triangleright \dot{c}, \dot{A} : X}{\Sigma \triangleright \dot{c} \Rightarrow \dot{A} : X}$$

$$, \frac{\Sigma \triangleright \Gamma : X \quad \Sigma \triangleright \Delta : X}{\Sigma \triangleright \Gamma, \Delta : X}$$

Si le jugement de typage $\Sigma \triangleright \dot{A} : X$ est valide, on dira que l'agent $\dot{A}$ protège les variables X.

Proposition 7.28. *Soient Γ un multiensemble d'agents, X un ensemble de variables et Σ un environnement.*

$$Si \ \ \mathcal{V}(\Gamma) \cap X = \emptyset \ \ alors \ \ \Sigma \triangleright \Gamma : X$$

Preuve. Le résultat se prouve par une induction sur la taille de Γ. □

7.5.2 Système de contraintes cohérent

Cette section restreint les questions de typage aux systèmes de contraintes.

Définition 7.29 (Système de contraintes cohérent)**.** Un axiome $c \Vdash d$ est *cohérent* avec le système de types, si pour tout environnement de typage Σ et tout ensemble de variables X on a :

$$\text{Si } \Sigma \triangleright c : X \ \ \text{alors} \ \ \Sigma \triangleright d : X$$

Un système de contraintes est *cohérent* avec le système de types si tous ses axiomes sont cohérents avec le système de types.

Proposition 7.30 (Préservation du typage par implication)**.** *Soit* Γ *un ensemble de contraintes de* $\dot{\mathcal{C}}$*,* $\dot{c}$ *une contrainte de* $\dot{C}$*,* Σ *un environnement et* X *un type. Sous l'hypothèse que le système de contraintes* $\mathcal{D}$ *est cohérent avec le système de types, on a :*

1. *Si* $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}$ *et* $\Sigma \triangleright \Gamma : X$ *alors* $\Sigma \triangleright \dot{c} : X$
2. *Si* $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c} \otimes \top$ *et* $\Sigma \triangleright \Gamma : X$ *alors* $\Sigma \triangleright \dot{c} : X$

Preuve. Le résultat se démontre par induction sur la preuve π de $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}$:

- π est un axiome logique : immédiat
- π est un axiome non-logique : Il y a deux sous-cas :
  - l'axiome est hérité de $\mathcal{D}$, auquel cas il suffit d'utiliser la cohérence de $\mathcal{D}$ avec le système de types.
  - l'axiome provient du schéma d'axiome pour l'égalité pour les contraintes de $\dot{\mathcal{M}}$, il est donc de la forme :

    $$\dot{m}(x_1, \ldots, x_n) \otimes x_1 = y_1 \otimes \cdots \otimes x_n = y_n \vdash_{\dot{\mathcal{C}}} \dot{m}(\dot{x}_1, \ldots, \dot{x}_n)$$

    Il suffit de constater que les égalités qui apparaissent à fauche imposent que si $x_i \in X$ alors $y_i \in X$.
  - π finit par une coupure, sans perte de généralité, d'une des formes :

    $$\frac{\Gamma \vdash_{\dot{\mathcal{C}}} \dot{d} \quad \overline{\dot{d} \vdash_{\dot{\mathcal{C}}} \dot{c}}}{\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}} \qquad \text{ou} \qquad \frac{\overline{\dot{e} \vdash_{\dot{\mathcal{C}}} \dot{d}} \quad \dot{d}, \Delta \vdash_{\dot{\mathcal{C}}} \dot{e}}{\dot{e}, \Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}}$$

    Le même raisonnement que celui fait quand π était un axiome permet d'inférer que si $\Sigma \triangleright \dot{d} : X$ alors $\Sigma \triangleright \dot{c} : X$, pour le cas de gauche et que si $\Sigma \triangleright \dot{e} : X$ alors $\Sigma \triangleright \dot{d} : X$ pour le cas de droite. Il suffit alors d'utiliser l'hypothèse d'induction.
  - π finit par une introduction à gauche du $\exists$.

    $$\frac{\Gamma, d \vdash_{\mathcal{C}} c}{\Gamma, \exists x.d \vdash_{\mathcal{C}} c} x \notin \mathcal{V}(\Gamma, c)$$

    Il suffit de constater que si $x \notin \mathcal{V}(\Gamma, c)$ alors $\Sigma \triangleright \Gamma, \exists x.A : X \setminus \{x\}$ et $\Sigma \triangleright c : X \setminus \{x\}$.
  - π finit par une introduction à droite du $\exists$.

    $$\frac{\Gamma \vdash_{\mathcal{C}} c[x \backslash t]}{\Gamma \vdash_{\mathcal{C}} \exists x.c}$$

    Par hypothèse d'induction, $\Sigma \triangleright c[x \backslash t] : X$. Puisque, sans perte de généralité, on sait que $x \notin X \cup \mathcal{V}(\Gamma)$ il possible de se restreindre à deux sous-cas : Soit $\mathcal{V}(t) \subset X$ auquel cas $\Sigma \triangleright c : X \cup \{x\}$; soit $\mathcal{V}(t) \cap X = \emptyset$ auquel cas $\Sigma \triangleright c : X$. □

Il est intéressant de constater que la cohérence de $\mathcal{D}$ avec le système de types subsume la condition de cohérence de l'égalité (définition 3.7).

Proposition 7.31. *Si $\mathcal{D}$ est cohérent avec le système de types alors $\mathcal{D}$ respecte la cohérence de l'égalité.*

Preuve. Ce résultat se prouve par un raisonnement par l'absurde. Supposons que $\mathcal{D}$ est cohérent avec le système de types mais qu'il n'est pas cohérent avec l'égalité. Il existe alors une contrainte $\dot{c}$ et deux variables syntaxiquement distinctes x et y telles que $\dot{c} \vdash_{\dot{\mathcal{C}}} x = y \otimes \top$ et $\{x, y\} \not\subset \mathcal{V}(\dot{c})$. Il y a deux cas :

- $x \notin \mathcal{V}(\dot{c})$: Posons $X = \{y\} \cup \mathcal{V}(\dot{c})$. On constate alors que pour n'importe quel environnement Σ, le jugement de typage $\Sigma \triangleright \dot{c} : X$ est valide alors que le jugement $\Sigma \triangleright (x = y) : X$ ne l'est pas ; ce qui contredit le lemme précédent.
- $y \notin \mathcal{V}(\dot{c})$: ce cas est symétrique au précédent. □

7.5.3 Préservation du typage

Cette section étend les résultats de la section précédente aux agents modulaires. À partir de maintenant le système de contraintes $\mathcal{D}$ est supposé cohérent avec le système de types.

Lemme 7.32. *Soit Γ un multiensemble d'agents modulaires, x une variable non libre dans Γ et Σ un environnement de typage.*

$$\Sigma \triangleright \Gamma : X \uplus \{x\} \quad \textit{si et seulement si} \quad \Sigma \triangleright \Gamma : X$$

Preuve. Le lemme se démontre par induction sur la preuve de $\Sigma \triangleright \Gamma : X$. □

Lemme 7.33 (lemme de substitution)**.**

1. *Si $\mathcal{V}(\vec{s}, \vec{t}) \subset X$ alors $\Sigma \triangleright \Gamma[\vec{x} \backslash \vec{s}] : X$ implique $\Sigma \triangleright \Gamma[\vec{x} \backslash \vec{t}] : X$.*
2. *Si $\mathcal{V}(\vec{s}, \vec{t}) \cap X = \emptyset$ alors $\Sigma \triangleright \Gamma[\vec{x} \backslash \vec{s}] : X$ implique $\Sigma \triangleright \Gamma[\vec{x} \backslash \vec{t}] : X$.*

Preuve. On procède par induction sur la preuve de $\Sigma \triangleright \Gamma : X \cup \mathcal{V}(\vec{x})$ □

Lemme 7.34 (lemme de substitution dans les gardes)**.** *Soit d une contrainte de $\mathcal{C}$, s_1, s_2 et t trois termes, $\vec{x}$ une séquence de variables quantifiables de d telle que $x \notin \mathcal{V}(t)$. Soient $\Sigma \triangleright d[x \backslash s_1]^t : X$ et $\Sigma \triangleright d[x \backslash s_2]^t : X$ deux jugements de typage valides.*

1. *Si $\mathcal{V}(s_1) \subset X$ alors $\mathcal{V}(s_2) \subset X$*
2. *Si $\mathcal{V}(s_2) \cap X = \emptyset$ alors $\mathcal{V}(s_2) \cap X = \emptyset$.*
3. *Si $\mathcal{V}(t) \cap X = \emptyset$ alors $\mathcal{V}(s_1) \cap X = \mathcal{V}(s_2) \cap X = \emptyset$.*

Preuve. Les deux premiers cas sont des corollaires du lemme précédent. Le dernier se démontre par induction sur la taille de d. □

Théorème 7.35 (Préservation du Typage)**.** *Soient Σ un environnement de typage et X un ensemble de variables. Soit $\langle Y; \dot{c}; \Gamma\rangle \xrightarrow{*} \langle Y'; \dot{c}'; \Gamma'\rangle$ une dérivation modulaire cohérente, qui n'introduit que des variables fraîches vis-à-vis de X et telle que $\dot{c}$ soit décomposable.*

$$\textit{Si}\ \ \Sigma \rhd \dot{c}, \Gamma : X \ \ \textit{alors il existe}\ X' \ \textit{tel que}\ X \subset X' \ \textit{et}\ \Sigma \rhd \dot{c}', \Gamma' : X'$$

Preuve. On procède par induction sur la longueur de la dérivation :

- Le cas de base est immédiat.
- Pour le cas de la règle *tell*, il suffit de constater que si $\Sigma \rhd (c, d, \Gamma) : X$ alors $\Sigma \rhd (c \otimes d, \Gamma) : X$.
- Pour la règle de *localisation* : on constate que si $\Sigma \rhd (c, \exists z.\dot{A}, \Gamma) : X \cup \{z\}$ alors $\Sigma \rhd (c, \exists z.\dot{A}, \Gamma) : X \cup \{z\}$ ou $\Sigma \rhd (c, \exists z.\dot{A}, \Gamma) : X$. On conclut en supposant, sans perte de généralité, que z n'est pas dans X.
- La dernière transition est obtenue par application de la règle *ask* :

$$\frac{c \vdash_{\dot{c}} \exists Y.(d^t[\vec{y}\backslash\vec{t}] \otimes e) \qquad Y \cap \mathcal{V}(\Gamma) = \emptyset}{\langle X; \dot{c}; \forall\vec{y}(d^t \to \dot{A}), \Gamma\rangle \longrightarrow \langle X \uplus Y; \dot{e}; \dot{A}[\vec{y}\backslash\vec{t}], \Gamma\rangle}$$

  avec $\mathcal{V}(t) \cap \mathcal{V}(\vec{y}) = \mathcal{V}(t) \cap Y = \emptyset$.
  On remarque tout d'abord que si $\Sigma \rhd (\dot{c}, \forall\vec{y}(d^t \to \dot{A}), \Gamma) : X$, alors $\Sigma \rhd \dot{c} : X$ et $\Sigma \rhd \forall\vec{y}(d^t \to \dot{A}) : X$. Ainsi grâce à la préservation du typage par l'implication, on déduit que $\Sigma \rhd \exists Y.d^t[\vec{y}\backslash\vec{t}] \otimes e : X$. Les règles d'introduction du $\exists$ (resp. du $\forall$) imposent qu'il existe un sous-ensemble Y_1 de Y (resp. un sous-ensemble Y_2 de $\mathcal{V}(\vec{y})$), tel que $\Sigma \rhd d^t[\vec{y}\backslash\vec{t}] \otimes e : (X \cup Y_1)$ (resp. $\Sigma \rhd d^t \to \dot{A} : (X \cup Y_2)$). Comme Y_1 et Y_2 sont frais dans la configuration de droite, il est possible d'appliquer e lemme 7.32 pour inférer $\Sigma \rhd (d^t[\vec{y}\backslash\vec{t}], \dot{e}, d^t, \dot{A}, \Gamma) : (X \cup Y_1 \cup Y_2)$. L'application des deux lemmes précédents permet alors de déduire que $\Sigma \rhd (\dot{A}[\vec{y}\backslash\vec{t}]) : (X \cup Y_1 \cup Y_2)$. Comme Y_2 est un ensemble de variables non libres dans la configuration de droite, on conclut, grâce au lemme 7.32, que $\Sigma \rhd (\dot{e}, \dot{A}[\vec{y}\backslash\vec{t}], \Gamma) : (X \cup Y_1)$.
- Le cas du *ask* persistant est identique au cas du *ask* non persistants. □

7.5.4 Accessibilité dans les agents typés

Cette section s'intéresse aux liens entre agents bien typés et accessibilité de variables.

Proposition 7.36. *Soit $\dot{c}$ une contrainte décomposable.*

$$Si \;\; \Sigma \triangleright \dot{c} : Y \;\; et \; Y \cap X = \emptyset \;\; alors \;\; \mathcal{A}_{\dot{c}}(X) \cap Y = \emptyset$$

Preuve. On procède par induction sur la taille de $\dot{c}$:

- $\dot{c}$ est une contrainte classique : ce cas se démontre par l'absurde, supposons donc que $x \in \mathcal{A}_{\dot{c}}(X) \cap Y$. Comme $\dot{c}$ est classique $\mathcal{A}_{\dot{c}}(X) = \mathcal{A}^u_{\dot{c}}(X)$. Soit $d \in \mathcal{D}$ une contrainte classique et y une variable telles que $\mathcal{V}(c) \cap \mathcal{V}(d) \subset X$ et $\dot{c}, d \vdash_{\dot{\mathcal{C}}} x = y \otimes \top$ (on sait qu'une telle contrainte existe par définition de l'accessibilité par unification). Soient $Z = \mathcal{V}(d) \setminus \mathcal{V}(c)$ et w une variable fraîche. On a alors aisément $c, \exists Z.(d \otimes w = y) \vdash_{\dot{\mathcal{C}}} x = w \otimes \top$ et $\Sigma \triangleright (c, \exists Z.(d \otimes w = y)) : Y$. Or $\Sigma \triangleright x = w : Y$ n'est pas un jugement de typage valide, ce qui contredit la proposition 7.30.
- $\dot{c} = \dot{d} \dot{\otimes} \dot{m}(s, \vec{t})$: Par hypothèse d'induction, $\mathcal{A}_{\dot{d}}(X) \cap Y = \emptyset$. Il suffit alors de constater que si $\mathcal{V}(s) \subset \mathcal{A}_{\dot{d}}(X)$ le typage implique que $\mathcal{V}(\vec{t}) \cap Y = \emptyset$ et que si $\mathcal{V}(s) \not\subset \mathcal{A}_{\dot{d}}(X) = \emptyset$ alors $\mathcal{A}_{\dot{d} \dot{\otimes} \dot{m}(s,\vec{t})}(X) = \mathcal{A}_{\dot{d}}(X)$.
- $\dot{c} = \exists y.\dot{d}$: il suffit d'utiliser l'hypothèse d'induction. □

Théorème 7.37 (Protection des variables dans les agents bien typés)**.** *Soit $\langle Y; \dot{c}; \Gamma, \Delta \rangle \xrightarrow{*} \langle Y \uplus Y'; \dot{c}'; \Gamma', \Delta' \rangle$ une dérivation modulaire cohérente telle que $\dot{c}$ soit décomposable et que Γ' et Δ' soient les réduits respectifs de Γ et Δ. Soit X un ensemble de variables tel que $X \subset Y$.*

$$Si \;\; \Sigma \triangleright (\dot{c}, \Delta) : X \;\; et \;\; \mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma)) \cap X = \emptyset \;\; alors \;\; \mathcal{A}_{\dot{c}'}(\mathcal{V}(\Gamma')) \cap X = \emptyset$$

Preuve. Le résultat se démontre par induction sur la longueur de la dérivation. Le cas de base est immédiat. Pour le cas inductif, on suppose que la première transition est de la forme :

$$\langle Y; \dot{c}; \Gamma, \Delta \rangle \longrightarrow \langle Y \uplus Y''; \dot{c}''; \Gamma'', \Delta'' \rangle$$

On remarque que, si $\mathcal{A}_{\dot{c}}(\mathcal{V}(\Gamma)) \cap X = \emptyset$, la croissance de $\mathcal{A}^1$ impose que $\mathcal{V}(\Gamma) \cap X = \emptyset$. On déduit ainsi, à l'aide du lemme précédent, que $\Sigma \triangleright (\dot{c}, \Gamma, \Delta) : X$. On peut, alors, utiliser la stabilité du typage, pour déduire qu'il existe un sous-ensemble de Y'', X'' supposé, sans perte de généralité, disjoint de $\mathcal{V}(\dot{c}, \Delta, \Gamma)$, tel que $\Sigma \triangleright (\dot{c}'', \Gamma'', \Delta'') : (X \uplus X'')$. On procède maintenant par cas sur la première transition :

- La transition réduit une *localisation* ou un *tell* dans Γ : Dans ce cas, il suffit de prendre $X'' = \emptyset$ et d'appliquer l'hypothèse d'induction sur la configuration de droite.

– La première transition réduit un *ask* dans Γ :

$$\frac{c \vdash_{\dot{\mathcal{C}}} \exists Y.(d^t[\vec{y}\backslash\vec{s}] \otimes e) \qquad Y'' \cap \mathcal{V}(\Gamma, \Delta) = \emptyset}{\langle Y;\, \dot{c};\, \forall \vec{y}(d^t \to A), \Gamma''', \Delta\rangle \longrightarrow \langle Y \uplus Y'';\, \dot{e};\, A[\vec{y}\backslash\vec{s}], \Gamma''', \Delta\rangle}$$

avec $\mathcal{V}(t) \cap \mathcal{V}(\vec{y}) = \mathcal{V}(t) \cap Y = \emptyset$.
Comme les variables de X ne sont pas libres dans Γ, $\mathcal{V}(t) \cap X = \emptyset$ et donc $\mathcal{V}(t) \cap (X \cup X'') = \emptyset$ (car X'' est un ensemble frais). Ainsi en utilisant le lemme de substitution dans les gardes, on déduit que $\mathcal{V}(\vec{s}) \cap (X \cup X'') = \emptyset$ et que donc $\mathcal{V}(\Gamma'') \cap (X \cup X'') = \emptyset$. On conclut le cas en appliquant l'hypothèse d'induction.

– Le cas d'une réduction d'un *ask persistant* dans Γ est analogue au cas précédent.

– La première transition réduit un agent de Δ : il suffit d'utiliser le lemme 7.36 et d'appliquer l'hypothèse d'induction. □

7.5.5 Protection du code dans les agents bien typés

Cette section s'intéresse aux liens entre agents bien typés et protection du code.

Lemme 7.38. *Soit Γ un multiensemble cohérent de contraintes de $\dot{\mathcal{C}}$, $\dot{m}$ une contrainte atomique de $\dot{\mathcal{M}}$, d une contrainte de $\mathcal{C}$ et x une variable arbitraire :*

(i) Si $\Gamma, \dot{m} \vdash_{\dot{\mathcal{C}}} d^x$ alors $x \in \mathcal{A}_\Gamma(\mathcal{V}(m))$.

(ii) Si $\Gamma, \dot{m} \vdash_{\dot{\mathcal{C}}} d^x \otimes \top$ et $\Gamma \not\vdash_{\dot{\mathcal{C}}} d^x \otimes \top$ alors $x \in \mathcal{A}_\Gamma(\mathcal{V}(m))$.

Preuve. (i) se démontre par induction sur la preuve π de $\Gamma, \dot{m} \vdash_{\dot{\mathcal{C}}} d^x$.

– π est un axiome logique : immédiat car $\dot{m} = d^x$ et $\mathcal{A}_1(\mathcal{V}(m)) = \mathcal{V}(m)$.

– π ne peut pas être un axiome non-logique car il n'existe pas d'axiome non-logique avec une contrainte atomique de $\dot{\mathcal{M}}$ à droite.

– π finit par une coupure : sans perte de généralité, cette coupure est de la forme :

$$\frac{\Gamma, \dot{m} \vdash_{\dot{\mathcal{C}}} \dot{c} \quad \overline{\dot{c} \Vdash_{\dot{\mathcal{C}}} d^x}}{\Gamma, \dot{m} \vdash_{\dot{\mathcal{C}}} d^x}$$

Dans ce cas, d^x est nécessairement une contrainte de synchronisation de $\dot{\mathcal{M}}$ par stabilité de $\dot{\mathcal{C}}$ sur le sous-langage de $\mathcal{D}$ (cf. définition 3.10). $\dot{c}$ est donc de la forme $(y{=}x) \otimes (\vec{z}_1{=}\vec{z}_2) \otimes \dot{n}(y, \vec{z}_1)$ et donc de la forme e^y avec $e = (y = x) \otimes (\vec{z}_1 = \vec{z}_2) \otimes n(\vec{z}_1)$. Par hypothèse d'induction, $y\mathcal{A}_\Gamma(\mathcal{V}(m))$ et comme on a $\Gamma \vdash_{\dot{\mathcal{C}}} x = y\otimes$, on infère que $x\mathcal{A}_\Gamma(\mathcal{V}(m))$.

– Les autres cas sont des conséquences immédiates de l'hypothèse d'induction.

Le cas (ii) se prouve aussi par induction sur $\Gamma, \dot{m} \vdash_{\dot{\mathcal{C}}} d^x \otimes \top$. Le seul cas intéressant est l'introduction à droite du $\otimes$ qui sépare le membre droit :

$$\frac{\Delta_1 \vdash_{\dot{\mathcal{C}}} d^x \quad \Delta_2 \top}{\Delta_1, \Delta_2 \vdash_{\dot{\mathcal{C}}} d^x \otimes \top}$$

Il suffit de constater que $\dot{m}$ est nécessairement dans Δ_1 (sinon $\Gamma \nvdash_{\dot{\mathcal{C}}} d^x \otimes \top$) et d'utiliser le cas précédent. □

Théorème 7.39 (Protection du code dans les agents bien typés)**.** *Soient A, B et C trois agents de MLCC. Soient x et y deux variables, Σ un environnement de typage et X un ensemble de variables contenant x.*

Si $\Sigma \triangleright (A^x \parallel \dot{B}^y) : X$ alors A^x est dans l'agent $(\exists X.(x\{A\} \parallel B))^y$.

Preuve. On note d'abord que l'interprétation de $\exists X.(x\{A\} \parallel B)$ dans y est la configuration $\exists X.(A^x \parallel B^y)$. Grâce à la proposition 3.31, on peut supposer, sans perte de généralité, que toute les dérivations de $\langle \emptyset;\ \mathbf{1};\ \exists X.(A^x \parallel B^y) \parallel C^t \rangle$ commencent par $\langle \emptyset;\ \mathbf{1};\ \exists X.(A^x \parallel B^y) \parallel C^t \rangle \xrightarrow{*} \langle X;\ \mathbf{1};\ A^x, B^y, C^t \rangle$ où $\mathcal{V}(C^t) \notin X$.

Supposons maintenant la dérivation suivante, où D_A est un réduit de A et m^t un réduit de C^t :

$$\langle X;\ \mathbf{1};\ A^x, B^y, C^t \rangle \xrightarrow{*} \langle Y;\ \dot{c};\ \Gamma, D_A, m^{t'} \rangle \longrightarrow \langle Y;\ \dot{c} \otimes m^t;\ \Gamma, D_A \rangle \longrightarrow \langle Y';\ \dot{c}';\ \Gamma, D'_A \rangle$$

On procède par cas sur le type de la dernière transition :

– La dernière transition est une *localisation* ou un *tell* : Il suffit d'utiliser le lemme de permutation (lemme 3.29).

– La dernière transition est un *ask* ou *ask* persistant : On constate que les gardes des *ask* (persistants ou non) de A^x et ses réduits sont nécessairement de la forme d^x (on rappelle que A est supposé sans module interne). En supposant, sans perte de généralité, que $(X) \cap \mathcal{V}(C^t)$, on a $\Sigma \triangleright A^x, B^y, C^t : X$. Grâce à la propriété de protection des variables (proposition précédente) on sait donc que $x \notin \mathcal{A}_{\dot{c}}(m^t)$. On conclut à l'aide du lemme précédent. □

7.6 Discussions

Dans la premier partie de cette thèse nous avons vu qu'il existait un lien fort entre variables logiques et canaux de communication. Dans le schéma

MLCC, un canal de communication est typiquement la référence d'un module (par exemple un terme t dans le cas où un agent de la forme $t\{A\}$ est défini). Dans ce cadre la gestion de tels canaux est particulièrement flexible. Un canal peut être global, c.-à-d. connu par tous (cas typique de l'utilisation d'une constante de Σ_F comme référence), mais aussi anonyme (cas de l'utilisation d'une variable comme référence). Comme cela a été démontré précédemment, un agent est capable de garder certains de ses canaux totalement privés, mais il est aussi tout à fait capable de les passer à un autre agent (c'est le cas de l'itérateur de liste cf. 7.8, où l'agent voulant utiliser l'itérateur passe la référence à un module contenant un *ask* persistant synchronisé sur contrainte $apply(x)$). Dans le cas où des agents s'échangent des canaux de communication, ceux-ci restent a priori connus seulement des modules concernés.

L'utilisation d'un terme quelconque comme référence de module permet même de communiquer une information partielle sur un canal de communication. Comme l'illustre l'exemple suivant, si un module est référencé par un terme contenant plusieurs variables distinctes, un module tiers ne pourra accéder à ce module que s'il possède toutes les variables de la référence.

Exemple 7.40. Soit un agent A_1 possédant un module interne B référencé par un triplet (z_1, z_2, z_3) de variables privées. Cet agent communique les variables z_1,z_2 et z_3 à un agent A_2 par l'intermédiaire respectif de trois agents I_1, I_2 et I_3. À l'autre bout de la chaîne, A_2, qui possède alors toutes les variables, est capable de faire appel au module B alors que les intermédiaires qui ne possèdent, a priori, qu'une seule variable sur les trois, ne le sont pas. La figure 7.1 illustre un tel échange. En supposant que les agents A_1, A_2, I_1, I_2 et I_3 sont référencés par les variables respectives x_1, x_2, y_1, y_2 et y_3, l'agent suivant est une instance possible de la figure.

$$
\begin{array}{l}
x_1\{\exists z_1 z_2 z_3.(A_1 \parallel (z_1, z_2, z_3)\{B\} \parallel x_{I_1} : m(z_1) \parallel x_{I_1} : m(z_1) \parallel x_{I_1} : m(z_1))\} \parallel \\
y_1\{\forall x(m(x) \Rightarrow I_1 \parallel x_2 : m_1(x))\} \parallel \\
y_2\{\forall x(m(x) \Rightarrow I_2 \parallel x_2 : m_2(x))\} \parallel \\
y_3\{\forall x(m(x) \Rightarrow I_3 \parallel x_2 : m_3(x))\} \parallel \\
x_2\{\exists x_B.(\forall z_1 z_2 z_3((m_1(z_1) \otimes m_2(z_2) \otimes m_3(z_3)) \Rightarrow x_B = (z_1, z_2, z_3)) \parallel A_2)\}
\end{array}
$$

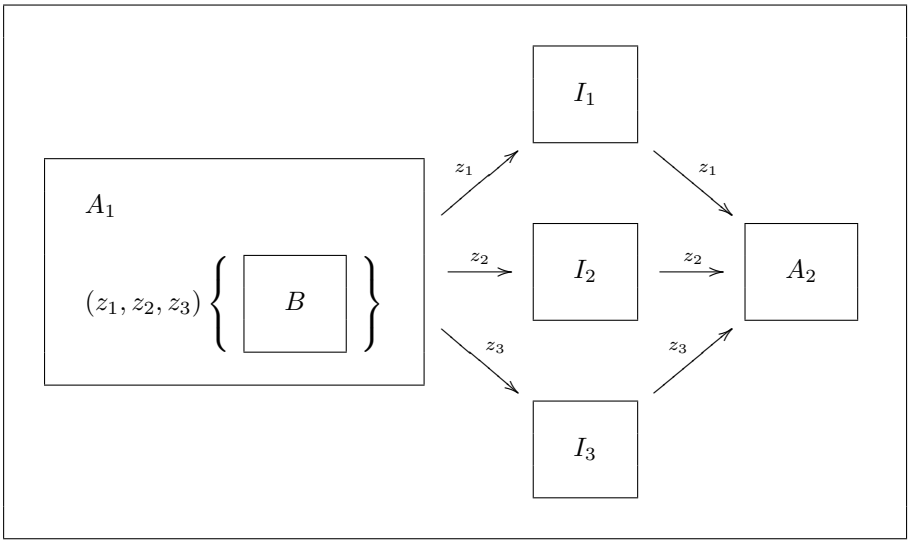

FIG. 7.1 – Echange d'une référence à l'aide de plusieurs intermédiaires

Chapitre 8

Équivalence Observationnelle dans MLCC

Sommaire

8.1 Introduction

Ce chapitre reprend la notion d'équivalence observationnelle présenté au chapitre 5. L'équivalence observationnelle est tout d'abord redéfinie au cadre modulaire dans la section 8.2. Nous montrons ensuite, dans la section 8.3 que l'utilisation d'observateurs moins généraux que ceux utilisés dans la définition ne modifie pas l'équivalence. Finalement nous comparons, à la section 8.5, cette équivalence avec d'autres équivalences classique.

Nous reprenons dans ce chapitre les notations du chapitre précédents. Notamment, $\mathcal{D}$ représentera le langage des contraintes classique, $\dot{\mathcal{M}}$ le langage

des contraintes de synchronisation auxquelles ont été ajoutées un argument un argument et $\dot{\mathcal{C}}$ le langage des contraintes modulaires.

8.2 Définition

Définition 8.1 (Équivalence Observationnelle)**.** Deux agents modulaires A_1 et A_2 sont observationnellement équivalents sur un sous-langage $\dot{\mathcal{C}}'$ de $\dot{\mathcal{C}}$ si pour tout contexte modulaire C, $\mathcal{O}^{\dot{\mathcal{C}}'} = \mathcal{O}^{\dot{\mathcal{C}}'}(C[A_2])$. Dans ce cas, on notera $A_1 \approx_{\dot{\mathcal{C}}'} A_2$.

8.3 Simplification des observateurs

8.3.1 Simplification des observateurs pour $\approx_{\dot{\mathcal{C}}}$

Dans le cas où l'observable est $\dot{\mathcal{C}}$, on retrouve facilement le théorème de simplification des observateurs (théorème 5.14). En effet comme l'énonce la proposition suivante, l'induction de la preuve correspondante reste valide au cadre modulaire.

Proposition 8.2. *Un séquent $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}$, tel que Γ est un multiensemble traductions d'agent modulaire (resp. de contraintes) et $\dot{c} \in \dot{\mathcal{C}}$ une contrainte, est prouvable si et seulement si $\Gamma \vdash_{\dot{\mathcal{C}}} \dot{c}$ admet une preuve dont tous les sous-arbres ont pour conclusion un séquent de la forme $\Delta \vdash_{\dot{\mathcal{C}}} \dot{d}$ où Δ est un multiensemble de traductions d'agents modulaires et $\dot{d} \in \dot{\mathcal{C}}$ une contrainte.*

Preuve. La preuve de cette proposition est pratiquement identique à celle de la proposition 3.35. □

Théorème 8.3. *Soient A_1 et A_2 deux agents modulaires. $A_1 \approx_{\dot{\mathcal{C}}} A_2$ si et seulement si $\mathcal{O}^{\dot{\mathcal{C}}}(B \parallel A_1) = \mathcal{O}^{\dot{\mathcal{C}}}(B \parallel A_2)$ pour tout agent modulaire B.*

Preuve. La preuve de ce théorème suit celle du théorème 5.14 et des lemmes 2.30 et 2.31 en remplaçant la proposition 3.35 par la proposition 8.2. □

La simplification des observateurs est donc encore possible dans le cadre des agents modulaire, par contre la propriété de simplification des observables (proposition 5.15) n'est plus vraie dans le contexte modulaire. En effet la protection du code rend inaccessibles certaines contraintes de synchronisation, à l'observateur.

Exemple 8.4. Soient $m \in \dot{\mathcal{M}}$ une contrainte de synchronisation, $d \in \mathcal{D}$ une contrainte cohérente et x et y deux variables. Sous ces hypothèses on a $(\exists y.y\{m \otimes d\})^x \approx_{\mathcal{D}} d$ mais pas $(\exists y.y\{m \otimes d\})^x \not\approx_{\dot{\mathcal{C}}} d$ et ce bien que $\mathcal{D}$ soit un sous-langage dense de $\dot{C}$ ($\mathcal{D}$ est dense car il contient l'égalité cf exemple 3.9).

La simplification des observables n'étant pas possible dans le cadre modulaire $\approx_{\dot{\mathcal{C}}}$ est à priori différent de $\approx_{\dot{\mathcal{C}}'}$ (pour $\dot{\mathcal{C}}'$ un sous-langage de $\dot{\mathcal{C}}$ donné). Il est donc nécessaire de prouver la propriété de simplification des observateurs pour chaque équivalence observationnelle. Nous nous intéressons maintenant à l'équivalence $\approx_{\mathcal{D}}$.

8.3.2 Simplification des observateurs pour $\approx_{\mathcal{D}}$

Lemme 8.5. *Soient A_1 et A_2 deux traductions logiques d'agents modulaires. Si*
$(\forall \Gamma \in \dot{\mathcal{C}}^*, \dot{c} \in \dot{\mathcal{C}}, d \in \mathcal{D}, \dot{m}_1 \in \dot{\mathcal{M}}.$
$\quad((\Gamma, A_1 \vdash_{\dot{\mathcal{C}}} \dot{c} \wedge \dot{c} \dashv\vdash_{\dot{\mathcal{C}}} d \otimes \dot{m}_1) \Rightarrow \exists \dot{m}_2 \in \dot{\mathcal{M}}.(\Gamma, A_2 \vdash_{\mathcal{C}} d \otimes \dot{m}_2))$
alors pour tout contexte C
$(\forall \Gamma \in \dot{\mathcal{C}}^*, \dot{c} \in \dot{\mathcal{C}}, d \in \mathcal{D}, \dot{m}_1 \in \dot{\mathcal{M}}.$
$\quad((\Gamma, C[A_1] \vdash_{\dot{\mathcal{C}}} \dot{c} \wedge \dot{c} \dashv\vdash_{\dot{\mathcal{C}}} d \otimes \dot{m}_1) \Rightarrow \exists \dot{m}_2 \in \dot{\mathcal{M}}.(\Gamma, C[A_2] \vdash_{\mathcal{C}} d \otimes \dot{m}_2))$.

Preuve. On procède tout d'abord par induction sur la taille du contexte C (induction principale). Les cas de base – où C est un trou ou un contexte sans trou – sont immédiats. Le cas inductif se traite par induction sur la la preuve π de $\Gamma, C[A_1] \vdash_{\dot{\mathcal{C}}} c$ (induction secondaire). Grâce à la proposition 8.2, on sait que cette induction.

- π est un axiome non-logique ou une coupure de formes respectives :

$$C[A_1] \vdash_{\dot{\mathcal{C}}} \dot{c}' \qquad \text{ou} \qquad \frac{C[A_1] \vdash_{\dot{\mathcal{C}}} \dot{c}' \quad \Gamma, \dot{c}' \vdash_{\mathcal{C}} \dot{c}}{C[A_1] \vdash_{\dot{\mathcal{C}}} \dot{c}}$$

  Il y a deux cas suivants l'origine de l'axiome :
  - $\dot{c}' \in \mathcal{D}$: Dans ce cas, $C[A_1]$ et A_1 sont des contraintes de $\mathcal{D}$. Par hypothèses, on a alors $A_2 \vdash_{\dot{\mathcal{C}}} A_1 \otimes \dot{m}_2$, pour une certaine $\dot{m}_2 \in \dot{\mathcal{M}}$. On conclut en constatant que puisque C ne contient que des $\otimes$ et des $\exists$, $C[A_2] \vdash_{\dot{\mathcal{C}}} C[A_1] \otimes \exists Y.\dot{m}_2$ pour un certain Y.
  - $\dot{c}' \in \dot{\mathcal{M}}$: Dans ce cas, l'axiome provient du schéma d'axiomes pour l'égalité et est donc de la forme $\dot{m}(\vec{x}) \otimes (\vec{x} = \vec{y}) \vdash_{\dot{\mathcal{C}}} \dot{m}(\vec{y})$. Il y a deux cas suivant la valeur de C :
    - $C = \dot{m}(x) \otimes [\,]$: Dans ce cas, $A_1 = (\vec{x} = \vec{y})$. Pour conclure on constate que par hypothèses on a $A_2 \vdash_{\dot{\mathcal{C}}} (\vec{x} = \vec{y}) \otimes \dot{m}_2$, pour une certaine $\dot{m}_2 \in \dot{\mathcal{M}}$.

- $C = [\,] \otimes (\vec{x} = \vec{y})$. Dans ce cas, $A_1 = \dot{m}(x)$. Démontrons tout d'abord, par un raisonnement par l'absurde que $A_2 \vdash_{\dot{\mathcal{C}}} \dot{m}(x) \otimes \top$: supposons donc que $A_2 \nvdash_{\dot{\mathcal{C}}} \dot{m}(x) \otimes \top$. Soit $d' \in \mathcal{D}$ tel que $A_2 \nvdash_{\dot{\mathcal{C}}} d' \otimes \top$ (on sait que cela est possible puisque $\mathcal{D}$ est dense sur $\dot{\mathcal{C}}$). Sous ces deux hypothèses $\dot{m}(x) \multimap d', A_1 \nvdash_{\dot{\mathcal{C}}} d$ et $\dot{m}(x) \multimap d', A_2 \nvdash_{\dot{\mathcal{C}}} d' \otimes \top$ ce qui contredit les hypothèses du lemme.

  On conclut le sous-cas, en constatant que puisque $\dot{m}(x) \vdash_{\dot{\mathcal{C}}} \mathbf{1} \otimes \dot{m}(x)$ et $A_2 \vdash_{\dot{\mathcal{C}}} \dot{m}(x) \otimes \top$, $A_2 \vdash_{\dot{\mathcal{C}}} \dot{m}(x) \otimes \top$, les hypothèses du lemme implique que $A_2 \vdash_{\dot{\mathcal{C}}} \mathbf{1} \otimes \dot{m}(x) \otimes \dot{m}_2$, pour une certaine $\dot{m}_2 \in \dot{\mathcal{M}}$.

- les autres cas sont identiques à ceux de la preuve du lemme 5.12. □

Lemme 8.6. *Soient A_1 et A_2 deux traductions logiques d'agent modulaire. Si pour tout multiensemble de formules Γ et toute contrainte $d \in \mathcal{D}$, $\Gamma, A_1 \vdash_{\mathcal{C}} d \otimes \top$ implique $\Gamma, A_2 \vdash_{\mathcal{C}} d \otimes \top$ alors pour tout multiensemble de formules Γ, tout contexte C et toute contrainte $d \in \mathcal{D}$, $\Gamma, C[A_1] \vdash_{\mathcal{C}} d \otimes \top$ implique $\Gamma, C[A_2] \vdash_{\mathcal{C}} d \otimes \top$*

Preuve. La preuve est identique à celle du lemme 5.13. □

Théorème 8.7. *Soient A_1 et A_2 deux agents modulaires. $A_1 \approx_{\mathcal{D}} A_2$ si et seulement si $\mathcal{O}^{\mathcal{D}}(B \parallel A_1) = \mathcal{O}^{\mathcal{D}}(B \parallel A_2)$ pour tout agent modulaire B.*

Preuve. Ce théorème est corollaire la semantique logique de LCC et des deux lemmes précédents. □

8.4 Simplification des observables

On montre qu'il est possible d'adapter en partie la proposition 5.15 au cadre modulaire en se restreignant au sous-langage dense du langage de contraintes classique ?

Proposition 8.8. *Soient A_1 et A_2 deux agents et $\mathcal{D}'$ un sous-langage dense du langage $\mathcal{D}$.*

$$A_1 \approx_{\mathcal{D}} A_2 \quad \textit{si et seulement si} \quad A_1 \approx_{\mathcal{D}'} A_2$$

Preuve. Puisque tous les arguments des contraintes de $\mathcal{D}$ sont quantifiables à l'aide de $\forall$, on peut utiliser la même preuve que celle de la proposition 5.15. □

Par contre, il n'est pas possible de restreindre arbitrairement les observables sur $\dot{\mathcal{C}}$. Par exemple, grâce à la propriété de protection du code présentée au chapitre précédent il est asse facile de démontrer que $\exists x.\dot{m}(x) \approx_{\mathcal{D}} \mathbf{1}$ alors que $\exists x.\dot{m}(x) \not\approx_{\dot{\mathcal{C}}} 1$ (en effet $(\exists x.\dot{m}(x)) \in \mathcal{O}^{\dot{\mathcal{C}}}(\exists x.\dot{m}(x))$ clairement alors que $(\exists x.\dot{m}(x)) \notin \mathcal{O}^{\dot{\mathcal{C}}}(\mathbf{1})$).

8.5 Comparaisons entre différentes Équivalences

8.5.1 Équivalence observationnelle et équivalence logique

Comme l'illustre l'exemple suivant les deux équivalences observationnelles sont deux notions distinctes.

Exemple 8.9. Soient les deux agents MLCC suivants :

- $A = \forall x(m(x) \Rightarrow x = 0)$
- $B = \exists y.(\forall x(m(x) \Rightarrow y{:}n(x)) \parallel y\{\forall x(n(x) \Rightarrow x = 0)\})$

Bien évidemment $A^z \not\approx_{\dot{\mathcal{C}}} B^z$ car $\exists y.\dot{n}(y, x)$ est une contrainte accessible pour $\dot{m}(z, x) \parallel B^z$ mais par pour $\dot{m}(z, x) \parallel A^z$. Néanmoins $A^z \approx_{\mathcal{D}} B^z$ car, grâce à la protection du code (proposition 7.26) aucun observateur ne peut se synchroniser sur la contrainte de synchronisation $\dot{n}(y, x)$ qui dérive de l'agent B, ainsi aucun observateur ne verra des différences entre A^z et b^z.

8.5.2 Équivalence observationnelle et β-équivalence

Encodage du λ-calcul

Comme dans la section 4.3, l'ensemble $\mathcal{V}_\lambda$ des variables du λ-calcul $\mathcal{V}_e$ ainsi qu'un ensemble dénombrable de variables d'encodages seront supposés disjoints et inclus dans l'ensemble $\mathcal{V}$ des variables *LCC*

Définition 8.10 (Codage du λ-calcul avec « appel par valeur » dans MLCC)**.**

Chaque terme M du λ-calcul est codé par $[\![M]\!]$ une fonction des variables vers les agents LCC et définie récursivement par :

$$[\![x]\!](y) \stackrel{def}{=} (x{=}y) \otimes value(y)$$
$$[\![\lambda x.M]\!](y) \stackrel{def}{=} y\{\forall xz(apply(x, z) \otimes value(x) \Rightarrow [\![M]\!](z)) \parallel value(y))\}$$
$$[\![MN]\!](y) \stackrel{def}{=} \exists zz'.(z{:}apply(z', y) \parallel [\![M]\!](z) \parallel [\![N]\!](z'))$$

où y, z et z' sont supposées appartenir à $\mathcal{V}_e$. De plus $\mathcal{C}$ est supposé être le système de contraintes construit à partir des axiomes non-logiques suivants :

1. $x{=}y \Vdash_{\mathcal{C}} !x{=}y$ (= est une contrainte classique)
2. $\Vdash_{\mathcal{C}} x{=}x$ (= est réflexive)
3. $x{=}y \Vdash_{\mathcal{C}} y{=}x$ (= est symétrique)
4. $x{=}y \otimes y{=}z \Vdash_{\mathcal{C}} x{=}z$ (= est transitive)
5. $value(x) \Vdash_{\mathcal{C}} !value(x)$ ($value$ est une contrainte classique)
6. $value(x) \otimes x{=}y \Vdash_{\mathcal{C}} value(y)$ (substitution pour $value$)
7. $apply(x,y,z) \otimes (x,y,z){=}(x',y',z') \Vdash_{\mathcal{C}} apply(x',y',z')$ (substitution pour $apply$)

Exemple 8.11. Le codage du λ-terme $M = (\lambda x.xx)y$ est l'agent :

$$
\begin{array}{l}
\exists z_1 z_2 (z_1 : apply(z_2, z_\epsilon) \parallel \\
\quad z_1\{\forall x z_{10}.(apply(x, z_{10}) \Rightarrow \\
\quad\quad \exists z_{101} z_{102}.(z_{101} : apply(z_{102}, z_{10}) \parallel \\
\quad\quad\quad x = z_{101} \otimes value(z_{101}) \parallel \quad \} M|_{101} = x \\
\quad\quad\quad x = z_{102} \otimes value(z_{102}) \quad \} M|_{102} = x \\
\quad\quad) \\
\quad)\} \parallel \\
\quad y = z_2 \otimes value(z_2) \\
)
\end{array}
\qquad
\left.\begin{array}{c} M|_{10} \\ = \\ (xx) \end{array}\right\} \left.\begin{array}{c} M|_1 \\ = \\ \lambda x.xx \end{array}\right\}
\quad \} M|_2 = y
$$

Correction et Complétude

Proposition 8.12 (Correction)**.** *Soient M et N deux λ-termes, $\mathcal{D}$ un sous-language quelconque de $\mathcal{C}$ et y une variable. Si $M \to_{\beta_v} N$ alors pour tout contexte modulaire C on a $\mathcal{O}^{\mathcal{D}}(C[[\![M]\!](y)]) \subset \mathcal{O}^{\mathcal{D}}(C[[\![N]\!](y)])$.*

Preuve. Aux vues des conventions syntaxiques de MLCC, présentées au chapitre précédent, cet encodage est identique à celui proposé pour LCC non-modulaire. Le résultat de correction logique (proposition 4.10) du chapitre 4 s'applique donc ici. □

Théorème 8.13. *Soient M et N deux λ-termes, $\mathcal{D}$ un sous-language quelconque de $\mathcal{C}$ et $y \in \mathcal{V}$ une variable.*

$$Si \;\; M \leftrightarrow_{\beta_v} N \;\; alors \;\; [\![M]\!](y) \sim_{\mathcal{D}} [\![N]\!](y).$$

Chapitre 9

Applications de MLCC : Un Système de Modules Paramétriques pour la PLC

Sommaire

9.1 Introduction

Le schéma MLCC, présenté dans le chapitre précédemment, est instancié en un système de modules pour PLC, appelé mCLP. Ce système peut être vu comme une extension, incluant des modules dynamiques, du système MPLC présenté dans le chapitre 6. Ce système est présenté ici, avec une sémantique logique en logique linéaire et une implémenté à l'aide de fermeture. Ce système de modules a été implémenté, comme une «preuve de concept», dans un prototype de compilateur vers C disponible en ligne à l'adresse `http://contraintes.inria.fr/~haemmerl/pub/mclp.tgz`.

9.2 Le langage mCLP

9.2.1 Conventions syntaxiques de mCLP

Nous adoptons pour mCLP, une syntaxe pragmatique proche de celle de celle des systèmes PLC classiques. Comme en Prolog, les identifiants commençant par une majuscule représente des variables.

Définition 9.1. La syntaxe des programmes mCLP est définie par la grammaire suivante qui distingue les déclarations `D` et les buts `G` :

```
G ::=  module(T,E){D} | T:p(S1,...,Sn| | p(S1,...,Sn) |
 c(S1,...,Sn) | G,G | G ;G

D ::=  p(S1,...,Sn) :- G.D | p(S1,...,Sn).D | :- G.D | ϵ
```

où `T` est une variable ou un atome, `E` une liste de variables, $\mathtt{S}_1$, ..., $\mathtt{S}_n$ une séquence de termes, `c` une contraintes de $\mathcal{D}$ et `p` un symbole de prédicat.

Une déclaration mCLP est soit une clause, soit un but de la forme `:-G.` exécuté à l'initialisation du module. En plus de la conjonction, disjonction et ajout de contraintes usuelles, le but `module(T,E){D}` dénote l'*instantiation* d'un module `T` avec l'*implémentation* `D` et l'*environnement* `E`. L'environnement est simplement une liste de variables dont la portée est l'ensemble des clauses de l'implémentation du module. Si `T` est une variable libre, le module résultant sera dit *anonyme*, alors que si `T` est un atome, le module sera dît *nommé*. Le but $\mathtt{T:p(S}_1\mathtt{, \ldots, S}_2\mathtt{)}$ dénote un *appel global* au prédicat `p/n` défini dans le module `T` ; il est distingué de l'*appel local* au prédicat `p/n` défini dans le module courant et noté simplement $\mathtt{p(S}_1\mathtt{, \ldots, S}_2\mathtt{)}$.

9.2.2 Interprétation de mCLP dans MLCC

Les clauses sont interprétées par des *ask* persistants, attendant la contrainte de synchronisation qui représente l'appel de procédure. L'environnement du module fournit une nouvelle fonctionnalité permettant de manipuler des variables globales au module.

Définition 9.2. Formellement, l'interprétation des buts et des déclarations mCLP en MLCC est définie comme $[\![\mathtt{G}]\!]^{\mathtt{T}}$ et $[\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$ où `T` est la référence du module courant et `E` l'environnent courant :

$$[\![\mathtt{G}_1,\mathtt{G}_2]\!]^{\mathtt{T}} = [\![\mathtt{G}_1]\!]^{\mathtt{T}} \parallel [\![\mathtt{G}_2]\!]^{\mathtt{T}} \qquad [\![\mathtt{P}]\!]^{\mathtt{T}} = \mathtt{T:P} \qquad [\![\mathtt{S:P}]\!]^{\mathtt{T}} = \mathtt{S:P}$$
$$[\![\mathtt{G}_1;\mathtt{G}_2]\!]^{\mathtt{T}} = [\![\mathtt{G}_1]\!]^{\mathtt{T}} + [\![\mathtt{G}_2]\!]^{\mathtt{T}} \qquad [\![\mathtt{C}]\!]^{\mathtt{T}} = \mathtt{T:(C)} \qquad [\![\mathtt{module(S,E)\{D\}}]\!]^{\mathtt{T}} = \mathtt{S}\{[\![\mathtt{D}]\!]^{\mathtt{S}}_{\mathtt{E}}\}$$

$$[\![\texttt{:- G.D}]\!]^{\texttt{T}}_{\texttt{E}} = \exists \bar{\texttt{Y}} [\![\texttt{G}]\!]^{\texttt{S}} \parallel [\![\texttt{D}]\!]^{\texttt{T}}_{\texttt{E}}$$
$$[\![\texttt{p}(\vec{\texttt{t}}) \texttt{.D}]\!]^{\texttt{T}}_{\texttt{E}} = \forall \vec{\texttt{X}} (\texttt{p}(\vec{\texttt{X}}) \Rightarrow \exists \bar{\texttt{Y}} [\![\vec{\texttt{X}} \texttt{=} \vec{\texttt{t}}]\!]^{\texttt{S}}) \parallel [\![\texttt{D}]\!]^{\texttt{T}}_{\texttt{E}}$$
$$[\![\texttt{p}(\vec{\texttt{t}}) \texttt{ :- G.D}]\!]^{\texttt{T}}_{\texttt{E}} = \forall \vec{\texttt{X}} (\texttt{p}(\vec{\texttt{X}}) \Rightarrow \exists \bar{\texttt{Y}} [\![\vec{\texttt{X}} = \vec{\texttt{t}}, \texttt{G}]\!]^{\texttt{S}}) \parallel [\![\texttt{D}]\!]^{\texttt{T}}_{\texttt{E}}$$

où $\vec{\texttt{X}}$ est une ensemble de variables fraîches et $\bar{\texttt{Y}} = \mathcal{V}(\vec{\texttt{t}}, \texttt{G}) \setminus \texttt{E}$.

Cette traduction est supposée fonctionner avec le système de contraintes linéaire $(\mathcal{CP}, \Vdash_{\mathcal{CP}})$ tel que $\Vdash_{\mathcal{CP}}$ est le plus petit ensemble respectant les conditions suivantes :

- Si $(\texttt{c} \Vdash_{\mathcal{C}} \texttt{d})$ alors $(\texttt{c} \Vdash_{\mathcal{CP}} \texttt{d})$.
- Pour tout symbole de contrainte $\texttt{c}$ $\left(\texttt{c}(\vec{\texttt{X}}) \Vdash_{\mathcal{CP}} \texttt{!c}(\vec{\texttt{X}})\right)$.
- Pour tout symbole de prédicat $\texttt{p}$ $\left(\texttt{p}(\vec{\texttt{X}}), \vec{\texttt{X}} \texttt{=} \vec{\texttt{Y}} \Vdash_{\mathcal{CP}} \texttt{p}(\vec{\texttt{Y}})\right)$.

On peut noter que toute traduction $[\![A]\!]^{\texttt{T}}_{\texttt{E}}$ est pseudo stable sur $\mathcal{C}$, ainsi tous les résultats du chapitre 3 peuvent être appliqués aux programmes mCLP.

En plus d'une sémantique logique premier ordre, cette traduction illustre comment mCLP peut être compilé en utilisant les techniques classiques de compilations Prolog. Typiquement un module est référencé par une variable spéciale à laquelle l'environnement et les procédures du module sont attachés comme des attributs (cf. sous-section 2.3.6). Un prédicat mCLP est alors implémenté, par un prédicat Prolog ayant un argument supplémentaire, hérité de la traduction de MLCC en LCC (définition 7.2), et servant à stocker la référence au module courant.

9.3 Techniques de Programmation en mCLP

9.3.1 Variables Globales

L'environnent des modules permet d'introduire la notion de variable *globale*, c.-à-d. de variable partagée par les différentes clauses du module. Cette construction peut être utilisée, par exemple pour éviter de passer un argument à un grand nombre de prédicats d'un même module. Cependant, ces variables sont toujours des variables logiques usuelles et backtrackantes, au contraire des variables globales de GNU-Prolog (cf. sous-section 2.3.2)

Le programme suivant illustre l'utilisation d'une variable globale `Depth` dans le but d'implémenter un méta-interpréteur Prolog avec une stratégie d'évaluation équitable à l'aide d'une recherche par approfondissements successifs [Sti88]. Chaque clause `H:-B` du programme méta-interprété, est supposée être représentée à l'aide d'un fait de la forme `clause(H,B)` afin d'émuler le prédicat ISO `clause/2` [Int95] ; le prédicat `for(I, Begin, End)` produit

un point de choix où `I` est unifié à un entier compris entre `Begin` et `End` (voir par exemple [Dia03]).

Exemple 9.3. (Recherche par approfondissements successifs)

```
:-module(iter_deep, [Depth]){

   solve(G,N):-
      for(Depth,1,N),
      print('Depth: '), print(Depth),nl,
      iterative_deepening(G,0).

   iterative_deepening(_,I) :-
      (I >= Depth),!,
      fail.

   iterative_deepening(true, I):-!.

   iterative_deepening(((A,B)),I) :-!,
      iterative_deepening(A,I),
      iterative_deepening(B,I).

   iterative_deepening(A,I) :-
      clause((A:-B)),
      eval(I+1,J),
      iterative_deepening(B,J).

   /* Database */

   clause((member(X,[_|T]):-member(X,T))).
   clause((member(X,[X|_]):-true)).
   clause((print(A):-true)):-print(A),nl.
}.
```

Par exemple, il est possible d'énumérer sans boucler toutes les listes contenant 1,2,3 obtenue par une recherche de profondeur maximale 5, en exécutant le but `iter_deep:solve((member(1,L),member(2,L), member(3,L), print(L), fail), 5).`

9.3.2 Masquage du Code

Comme précédemment, on peut utiliser l'environnent pour créé une variable globale à un module, mais cette fois, cette variable est utilisée pour

garder caché, vis à vis de l'extérieur, module interne anonyme.

Ceci est illustré par le programme suivant qui fournit le prédicat `sort` utilisant une implémentation cachée du prédicat `quicksort`.

Exemple 9.4. (Quicksort) :

```
:- module(sort, [Impl]){

    sort(List, SortedList) :-
             Impl:quicksort(List, SortedList).

    :- module(Impl, []){
            quicksort([],[]).
            quicksort([X|Tail], Sorted) :-
                split(X, Tail, Small, Big),
                quicksort(Small, SortedSmall),
                quicksort(Big, SortedBig),
                append(SortedSmall, [X|SortedBig], Sorted).

                split(X, [],[],[]).
                split(X, [Y|Tail], [Y|Small], Big) :-
                    (X>Y), !,
                    split(X, Tail, Small, Big).
                split(X, [Y|Tail], Small, [Y|Big]) :-
                    split(X, Tail, Small, Big).

      }.
}.
```

La protection du code (c.f. section 7.4) assure qu'aucun appel aux prédicats `quicksort` et `split` n'est possible hors du module `sort`. L'exécution, en parallèle avec ce module, du but

`?- L=[1, 2/3, 5, 4/3, 1/2, 2/7], sort:sort(L, L1), print(L1), nl.`

affiche à l'écran la liste triée

`[2/7,1/2,2/3,1,4/3,5].`

9.3.3 Fermetures

La notion de fermeture présentée dans l'exemple 7.8 à travers la définition de module ayant, par convention, un prédicat `apply/1`. Cela rend possible la définition itérateurs pour les structures de données telles que `forall` et `exists` pour les listes, passant la fermeture comme un argument :

Exemple 9.5 (Itérateurs de listes). :

```
:- module(iterator, []){

   forall([], _).
   forall([H|T], C) :- C:apply(H), forall(T, C).

   exists([H|_], C) :- C:apply(H).
   exists([_|T], C) :- exists(T, C).

}.
```

En supposant que l'on dispose la contrainte de domaines finis =<, le prédicat built-in usuel `domain/3` (où `fd_domain/3` dans GNU-Prolog [Dia03]) d'un solveur de domaines finis peut être implémenté en utilisant l'opérateur de liste comme suit.

```
fd_domain(Vars, Min, Max):-
   module(Cl , [Min, Max]){
      apply(X) :- Min=<X , X=<Max.
   },
   ( list(Vars) -> iterator:forall(Vars, Cl) ;
     var(Vars) -> Cl:apply(Vars) ).
```

9.3.4 Modules Paramétriques

Les mécanismes de paramétrisation augmentent grandement les possibilités du programmeur en ce qui concerne la réutilisation de code. En effet, ils permettent de rendre l'implémentation de certains modules dépendant d'autres modules et de factoriser ainsi l'écriture de code. En combinant l'utilisation d'environnement aux mécanismes de paramétrisation des fermetures et du masquage de code présenté précédemment, il est possible d'obtenir un module dont l'implémentation est masquée et paramétrée par un module tiers.

L'exemple suivant montre comment paramétrer le module `sort` de l'exemple 9.4, en construisant un prédicat `generic_sort/2` qui crée dynamiquement des modules de tris (sont premier arguments) en utilisant un module passé en premier argument et supposé contenir un prédicat de comparaison >=.

Exemple 9.6. (Quicksort Paramétrique) :

```
:- module(sort, []){

    generic_sort(Sort, Order) :- module(Sort, [Order,Impl]){

        quicksort(List, SortedList) :-
            Impl:quicksort(List, SortedList).

        :- module(Impl, [Order]){
            quicksort([],[]).
            quicksort([X|Tail], Sorted) :-
                split(X, Tail, Small, Big),
                quicksort(Small, SortedSmall),
                quicksort(Big, SortedBig),
                append(SortedSmall, [X|SortedBig], Sorted).

                split(X, [],[],[]).
                split(X, [Y|Tail], [Y|Small], Big) :-
                    Order:(X>=Y), !,
                    split(X, Tail, Small, Big).
                split(X, [Y|Tail], Small, [Y|Big]) :-
                    split(X, Tail, Small, Big).

        }.

    }.
}.
```

En supposant qu'il existe deux modules `math` et `term` implémentant chacun les prédicats ISO [Int95] `>=` and `@>=`, l'exécution du but suivant affichera à l'écran les deux listes triées `[2/7,1/2,2/3,1,4/3,5]` et `[1,5,1/2,2/3,2/7,4/3]` :

```
L=[1, 2/3, 5, 4/3, 1/2, 2/7],
sort:generic_sort(Sort1, math), Sort1:sort(L, L1), print(L1), nl,
module(OrderLex, []){ X >= Y:- term:(X @>= Y) },
sort:generic_sort(Sort2, OrderLex), Sort2:sort(L, L2) print(L2), nl.
```

Chapitre 10

Conclusion et Perspectives

Notre travail a porté principalement sur l'étude de la modularité dans le cadre des langages concurrents avec contraintes fondés sur la logique linéaire (LCC). Cette étude s'est décomposée en deux parties. La première s'est concentrée sur l'internalisation d'un mécanisme de fermeture dans LCC. La seconde a portée sur l'ajout d'un système de modules simple et puissant.

La première partie de cette thèse s'est portée plus particulièrement sur l'internalisation des déclarations de procédure par l'agent *ask* persistant. Nous avons montré que ceci permettait d'encoder des mécanismes de fermetures, c.-à-d. la possibilité de manipuler du code avec un environnement comme des objets de première classe. Cette extension, qui reste premier ordre, a été obtenue par l'ajout d'une version persistante des agents *ask* logiquement équivalente à un *ask* bangué. Nous avons démontré que malgré l'ajout de l'opérateur bang dans les *asks* persistants, il était possible d'étendre, les résultats de sémantique logique de [FRS01].

Afin d'illustrer le pouvoir expressif du langage ainsi obtenu, nous avons proposé aux chapitres 4 et 5 des encodages compositionnels du λ-calcul et du π-calcul asynchrone.

La seconde partie de cette thèse s'est focalisée sur l'étude de la modularité dans les langages LCC. Une étude préliminaire de la modularité dans le cadre restreint des langages PLC, entreprise dans le chapitre 6, a mis en lumière les problèmes de protection du code en présence de méta-programmation. Nous avons montré comment l'utilisation de fermetures résout élégamment cette difficulté.

Dans le chapitre 7, un système de modules pour les langages LCC a ensuite été défini comme une simple couche de sucre syntaxique contraignant l'ensemble des agents définissables. En permettant de référencer les modules

par des variables, le langage résultant appelé MLCC (pour *Modular* LCC) internalise les modules comme des agents. Nous avons démontré que dans ce cadre l'accès à certaines variables pouvait être finement contrôlé, permettant ainsi de masquer et protéger du code grâce au quantificateur $\exists$.

Dans le chapitre 8, les résultats des chapitres 4 et 5 ont été adaptés au cadre MLCC. Nous avons ainsi démontré que MLCC permettait de définir une notion d'équivalence observationnelle plus abstraite que celle proposée en première partie, car ne distinguant pas les transitions interne aux modules. Finalement le pouvoir expressif du système de modules a été illustré, au chapitre 9, avec l'instanciation du schéma MLCC aux programmes logiques avec contraintes. Cela a conduit à un système de modules pour la PLC à la fois simple et puissant. Ce système de modules, qui supporte le masquage de code, les fermetures, la paramétrisation de modules a été implémenté dans un prototype.

Le système de modules que nous avons proposé se rapproche d'une certaine façon des approches logiques pour la conception de modules puisqu'il s'appuie sur la logique sous-jacente. Néanmoins il est de ce point de vue original car il reste totalement premier ordre alors que les autres approches logiques ont besoin de mécanismes d'ordre supérieur. Par exemple les propositions de Cabeza et al. [CHL04] et de Chen [Che87] sont basées sur des logiques d'ordre supérieur, celle de Brogi et al. [BMPT92] utilise une méta-logique et même celle de Miller [Mil94], qui a priori reste premier ordre, a besoin des méta-appels (c.-à-d. le `call`) pour la paramétrisation.

Ce système de modules partage avec ceux proposés dans les approches algébriques ou fonctionnelles sa grande expressivité. En effet, la notion de modules paramétriques que nous avons présentée à la section 9.3.4 reproduit les mécanismes d'abstraction de O'Keefe [O'K85] ou de foncteurs de Sannella et Wallen [SW92] (et plus généralement ceux des langages fonctionnels de type Standard ML [MTHM97] dont ils sont la transposition dans le cadre de la programmation logique).

Notre approche se distingue cependant des langages de programmation courants en ce qui concerne l'encapsulation du code. En effet l'encapsulation est généralement obtenue par des mécanismes proches commes les interfaces (langages objets tels que Smalltalk [GR83]), systèmes syntaxiques de module pour Prolog comme Ciao [BGC$^+$04]) et les signatures (langages fonctionnels de type Standard ML [GMM$^+$78] et Oz [MMR95]). Ici, nous avons utilisé l'opérateur $\exists$ pour contrôler la portée à la fois des données et du code.

Nous avons ainsi montré que LCC était un langage avec une sémantique opérationnelle très simple (qui se résume en 4 règles de réduction) et mono paradigme qui incorpore un grand nombres de traits de programmation

(contraintes, affectation impérative, modification dynamique de programmes, concurrence, fermetures ect.) mais aussi la modularité (c.-à-d. encapsulation et paramétrisation). Cette grande simplicité permet d'obtenir de façon directe une sémantique logique premier ordre pour tous ces traits de programmation.

Le problème de protection des variables privées dû au quantificateur universel qui est capable de contourner la restriction imposée par l'opérateur de localisation n'est pas propre à LCC. En effet bien qu'il ne soit pas absolument indispensable dans la version classique de CC avec déclarations, l'opérateur ($\forall$) est nécessaire pour encoder les fermetures par des implications logiques. Nous pensons donc que l'approche que nous avons proposée pour internaliser les modules à un langage de programmation est d'une portée assez générale pour les langages manipulant des variables logiques. Par exemple, le portage du système de modules présenté ici au dessus des *Constraint Handling Rules* (CHR) [Frü98] – qui possède par ailleurs une sémantique en logique linéaire [BF05] – ou des différentes classes de langages logiques concurrents comme ceux présentés par Shapiro dans [Sha89] ne semble pas poser de difficultés supplémentaire.

Nos travaux ont aussi permis d'approfondir l'étude des liens entre les canaux de communication du π-calcul et les variables logiques des langages LCC. Comme l'avait déjà remarqué Laneve et Montanari [LM92] dans les langages CC, les variables logiques partagent avec les canaux de communication la faculté de mobilité. Soliman a aussi montré [Sol04] que les canaux de communication pouvaient être encodés par des variables logiques alors que l'inverse était faux, notamment à cause de l'unification. Dans cette thèse, nous avons montré que l'unification permettait de connecter des canaux/variables sans présupposer dans quel sens circulait l'information à travers ces canaux et donnant ainsi aux variables un aspect réversible que n'avait pas le canaux de communication. Il est ainsi plus facile d'encoder dans les langages CC des relations symétriques entre agents.

La perspective immédiate ce cette thèse, est l'implémentation effective du système SiLCC (pour «*SiLCC is Linear logic Concurrent Constraint programming*») [Fag04, Hae05] qui a démarré au dessus d'une version de GNU Prolog que nous avons au préalable étendue avec les variables attribuées [DH05]. Ce projet a pour objectif principal la conception d'un langage riche combinant contraintes, concurrence et changement d'état. Ce système sera complètement *bootstrappé* au dessus d'un petit noyau LCC et fourni avec un grand nombre de bibliothèques, ce qui est actuellement le cas d'aucun système de PLC. Il pourra ainsi être facilement étendu par le programmeur lui-même. De ce point de vue, SiLCC a des préocupations proches de système de

programmation Oz/Mozart [MMR95], mais dans un cadre mono-paradigme, doté d'une sémantique logique.

Dans le système SiLCC, les solveurs de contraintes complexes seront directement écrits comme des bibliothèques. Tout solveur de contraintes devant être déterministe, il sera alors nécessaire de se poser des questions sur la confluence de tels programmes, comme cela à pu être fait pour les langages CC [MFP97] ou les CHR [Abd97]. De ce point de vue le développement d'analyseurs de confluence pour SiLCC peut s'appuyer sur la notion de paires critiques abstraites [HF07] que nous avons introduites en parallèle des travaux présentés ici, permettra le développement d'analyseurs de confluence pour SiLCC.

Une autre perspective naturelle de cette thèse est la réalisation d'une extension orientée objet pour les langages LCC. En effet, les travaux que nous avons mener ici aborde un certain nombre de points important pour la programmation orienté objet. Tout d'abord, l'utilisation de *ask* persistants ajoute aux agent CC, la possibilité de manipuler des d'états internes. Ensuite, le système de modules permet de garantir l'encapsulation des données. Finalement il est possible de spécialiser un module «parent» en lui ajoutant des clause à l'aide de l'unification de variables de module – qui réalise formellement l'union des déclarations des modules unifiés – introduisant ainsi une notion simple d'héritage. Il ne reste, en fait, qu'à définir un mécanise de portée pour la surcharge ou le masquage de portions de code nécessaire à la gestion de la liaison tardive.

Enfin, la suite du chapitre 7 est le développement d'un système de types complet pour MLCC en reprenant par exemple les résultats de [FC01, Coq04]. En plus de la possibilité de détecter des erreurs de programmation, le typage paramétrique devrait permettre de définir une classe d'agents plus riche que celle proposée en fin de ce chapitre.

Annexe A

Example d'internalisation des Modules

On se propose d'illustrer l'internalisation des modules dans deux paradigmes distincts. Pour cela nous construise un module `list` contenant trois procédures ou fonctions :

- `length` compte la taille d'une liste
- `reverse` inverse une liste
- `iter` qui applique à tout les arguements d'une liste une procédure passée en argument.

Internalisation des module dans un langage fonctionnel

L'internalisation de module dans le langage noyau d'ocmal[Ler07] consiste à voir un module comme un *record* dont les champs contiennent les fonctions du dît module. Dans cette internalisation, l'encapsulation des fonction interne au module est réaliser par la construction (`let ... in`).

Exemple A.1 (Le module `list` internalisé en ocaml).

```
type s = {
  length : 'a. 'a list -> int;
  reverse : 'a. 'a list -> 'a list;
  iter : 'a 'b. ('a -> unit) -> 'a list -> unit
}

let list = {
```

```
  length = (fun l ->
    let rec length_aux l n = match l with
      | [] -> n
      | h::t -> length_aux t (n+1)
    in  length_aux l 0)
  reverse = (fun l ->
    let rec reverse_aux l1 l2 = match l1 with
      | [] -> l2
      | h::t -> reverse_aux t (h :: l2)
    in reverse_aux l []
  );
  iter = ( fun f l ->
    let rec iter_aux f l = match l with
      | [] -> ()
      | h::t -> (f h)::(iter_aux f t)
    in iter_aux f l
  )
}
```

Il existe d'autres moyens d'internaliser les modules dans des langages fonctionnels, ces moyens étant plus ou moins limité par le système de typage du langage.

Internalisation des modules en CLP pure

L'internalisation suivante consiste à voir un module comme un predicat dont le premièr argument indexe les predicats du dît module.

Exemple A.2 (le module `list` internalisé en CLP).

```
list(length, [], 0).
list(length, [H|T], N):- list(length, T, N+1).
list(reverse, L1, L2):-list(reverse, L1, [], L).
list(reverse_aux, [], L, L).
list(reverse_aux, [H|T], A, L):-list(reverse_aux, T, [H|A], L).
```

L'implémentation de `iter` n'est pas possible étant donné qu'il n'exitse pas de moyen de faire de l'ordre supérieur en CLP pure. On note aussi que se système n'offre pas d'encapsulation stricte : le prédicat `reverse_aux` est toujours visible de l'exterieur du module.

Annexe B

Fragment Multiplicatif de la Logique Linéaire Intuitionniste

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

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

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

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

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

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

Le calcul des séquents est donné par les règles suivantes :

Axiome - Coupure

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

Multiplicatifs

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

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

Constantes

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

Bang

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

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

Quantificateurs

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

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

Annexe C

Séquents ILL Prouvés Automatiquement

Les séquents ILL suivants ont été prouvés automatiquement à l'aide du prouveur automatiquement llprover disponible en ligne à l'adresse `http://bach.istc.kobe-u.ac.jp/llprover/`.

Lemme C.1. $C \multimap D \multimap A \dashv\vdash C \otimes D \multimap A$

Preuve. Commande llprover : `c -> d -> a <--> c*d -> a`

$$\dfrac{\dfrac{\dfrac{\dfrac{}{C \vdash C}\,x \quad \dfrac{\dfrac{}{D \vdash D}\,x \quad \dfrac{}{A \vdash A}\,x}{D, D \multimap A \vdash A}\,\multimap\text{-l}}{C, D, C \multimap D \multimap A \vdash A}\,\multimap\text{-l}}{C \otimes D, C \multimap D \multimap A \vdash A}\,\otimes\text{-l}}{C \multimap D \multimap A \vdash C \otimes D \multimap A}\,\multimap\text{-r}$$

CPU Time = 0 msec.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{C \vdash C}\,x \quad \dfrac{}{D \vdash D}\,x}{D, C \vdash C \otimes D}\,\otimes\text{-r} \quad \dfrac{}{A \vdash A}\,x}{D, C, C \otimes D \multimap A \vdash A}\,\multimap\text{-l}}{C, C \otimes D \multimap A \vdash D \multimap A}\,\multimap\text{-r}}{C \otimes D \multimap A \vdash C \multimap D \multimap A}\,\multimap\text{-r}$$

CPU Time = 0 msec.

□

Lemme C.2. $!C \multimap D \otimes !C \multimap A \otimes !C \dashv\vdash !C \multimap D \multimap A$

Preuve. Commande llprover : `(!c) -> (d*!c) -> a*!c <--> !c->d -> a`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{!C \vdash !C}\,x \quad \dfrac{\dfrac{\dfrac{}{D \vdash D}\,x \quad \dfrac{}{!C \vdash !C}\,x}{D, !C \vdash D \otimes !C}\,\otimes\text{-r} \quad \dfrac{\dfrac{\dfrac{}{A \vdash A}\,x}{A, !C \vdash A}\,!\text{-w}}{A \otimes !C \vdash A}\,\otimes\text{-l}}{D, !C, D \otimes !C \multimap A \otimes !C \vdash A}\,\multimap\text{-l}}{D, !C, !C, !C \multimap D \otimes !C \multimap A \otimes !C \vdash A}\,\multimap\text{-l}}{D, !C, !C \multimap D \otimes !C \multimap A \otimes !C \vdash A}\,!\text{-c}}{!C, !C \multimap D \otimes !C \multimap A \otimes !C \vdash D \multimap A}\,\multimap\text{-r}}{!C \multimap D \otimes !C \multimap A \otimes !C \vdash !C \multimap D \multimap A}\,\multimap\text{-r}$$

CPU Time = 10 msec.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{!C \vdash !C}\,x \quad \dfrac{\dfrac{}{D \vdash D}\,x \quad \dfrac{}{A \vdash A}\,x}{D, D \multimap A \vdash A}\,\multimap\text{-l}}{D, !C, !C \multimap D \multimap A \vdash A}\,\multimap\text{-l} \quad \dfrac{}{!C \vdash !C}\,x}{D, !C, !C, !C \multimap D \multimap A \vdash A \otimes !C}\,\otimes\text{-r}}{D \otimes !C, !C, !C \multimap D \multimap A \vdash A \otimes !C}\,\otimes\text{-l}}{!C, !C \multimap D \multimap A \vdash D \otimes !C \multimap A \otimes !C}\,\multimap\text{-r}}{!C \multimap D \multimap A \vdash !C \multimap D \otimes !C \multimap A \otimes !C}\,\multimap\text{-r}$$

CPU Time = 30 msec. □

Lemme C.3. $(\mathbf{0} \multimap A) \otimes \top \dashv\vdash \mathbf{1} \otimes \top$

Preuve. Commande llprover : `bot -> a * top <--> 1 * top` [1]

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1}\text{-r} \quad \dfrac{}{\mathbf{0} \multimap A, \top \vdash \top}\,\top\text{-r}}{\mathbf{0} \multimap A, \top \vdash \mathbf{1} \otimes \top}\,\otimes\text{-r}}{(\mathbf{0} \multimap A) \otimes \top \vdash \mathbf{1} \otimes \top}\,\otimes\text{-l}$$

CPU Time = 0 msec.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0}, \top \vdash A}\,\mathbf{0}\text{-l}}{\top \vdash \mathbf{0} \multimap A}\,\multimap\text{-r} \quad \dfrac{}{\vdash \top}\,\top\text{-r}}{\top \vdash (\mathbf{0} \multimap A) \otimes \top}\,\otimes\text{-r}}{\mathbf{1}, \top \vdash (\mathbf{0} \multimap A) \otimes \top}\,\mathbf{1}\text{-l}}{\mathbf{1} \otimes \top \vdash (\mathbf{0} \multimap A) \otimes \top}\,\otimes\text{-l}$$

CPU Time = 0 msec. □

[1] llprover utilise la notation de Troelstra [Tro92] qui inverse $\bot$ et $\mathbf{0}$ par rapport à la notation originale de Girard. Les arbres de preuve utilisent la notation originale grâce à une redéfinition des macros LaTeX.

Lemme C.4. *Si* $c \vdash_{\mathcal{C}} d \otimes \top$ *alors* $(c \multimap d) \otimes \top \dashv\vdash_{\mathcal{C}} \mathbf{1} \otimes \top$.

Preuve. Commande llprover : `axioms([c->d]) & (c->d)*top<-->1*top.`

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1}\text{-r} \quad \dfrac{}{C \multimap D, \top \vdash \top}\,\top\text{-r}}{C \multimap D, \top \vdash \mathbf{1} \otimes \top}\,\otimes\text{-r}}{(C \multimap D) \otimes \top \vdash \mathbf{1} \otimes \top}\,\otimes\text{-l}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash C \multimap D}\,\mathrm{H} \quad \dfrac{}{\top \vdash \top}\,x}{\top \vdash (C \multimap D) \otimes \top}\,\otimes\text{-r}}{\mathbf{1}, \top \vdash (C \multimap D) \otimes \top}\,\mathbf{1}\text{-l}}{\mathbf{1} \otimes \top \vdash (C \multimap D) \otimes \top}\,\otimes\text{-l}$$

□

Bibliographie

[Aa01] Abderrahamane Aggoun and al. *ECLiPSe User Manual Release 5.2*, 1993 – 2001.

[Abd97] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266, Linz, 1997. Springer-Verlag.

[AD03] Salvador Abreu and Daniel Diaz. Objective : in minimum context. In *Proceedings of ICLP'2003, International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. Logic Programming. MIT Press, 1991.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.

[BCDP05] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraints catalog. Technical Report T2005-6, Swedish Institute of Computer Science, 2005.

[BdBP97] Eike Best, Frank S. de Boer, and Catuscia Palamidessi. Concurrent constraint programming with information removal. In *Proceedings of Coordination*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[BF05] Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In *Proceeding of CP 2005, 11th*, pages 137–151, 2005.

[BGC+04] F. Bueno, D. Cabeza Gras, M. Carro, M. V. Hermenegildo, P. Lopez-Garca, and G. Puebla. The ciao Prolog system. reference manual. Technical Report CLIP 3/97-1.10#5, University of Madrid, 1997-2004.

[BLM94] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programmming*, 19/20 :443–502, 1994.

[BMPT92] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In *META-92 : Third International Workshop on Meta Programming in Logic*, pages 105–119, Berlin, Heidelberg, 1992. Springer-Verlag.

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

[Cab04] Daniel Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid, August 2004.

[Car87] Mats Carlsson. Freeze, indexing, and other implementation issues in the wam. In Jean-Louis Lassez, editor, *Proceedings of ICLP'87, International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 40–58, Melbourne, 1987.

[CDKM95] D. Chemla, D. Diaz, P. Kerlirzin, and S. Manchon. Using clp(FD) to support air traffic flow management. In *3rd lnternational Conference on the Practical Applications of Prolog : PAP'95*, 1995.

[CFS93] Philippe Codognet, François Fages, and Thierry Sola. A metalevel compiler of clp(fd) and its combination with intelligent backtracking. *Constraint logic programming : selected research*, pages 437–456, 1993.

[CH00] Daniel Cabeza and Manuel Hermenegildo. A new module system for Prolog. In *First International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 131–148. Springer-Verlag, July 2000.

[Che87] Weidong Chen. A theory of modules based on second-order logic. In *The fourth IEEE. Internatal Symposium on Logic Programming*, pages 24–33, 1987.

[CHL04] Daniel Cabeza, Manuel Hermenegildo, and James Lipton. Hiord : A type-free higher-order logic programming langugae with predicate abstraction. In *Proceedings of ASIAN'04, Asian Computing Science Conference*, pages 93–108, Chiang Mai, 2004. Springer-Verlag.

[Col82] A. Colmerauer. Prolog ii manuel de référence et modèle théorique. Rapport Technique ERA CNRS 363, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, 1982.

[Col87] Alain Colmerauer. Opening the prolog iii universe. *BYTE*, 12(9) :177–182, 1987.

[Coq04] Emmanuel Coquery. *Typage et programmation en logique avec contraintes*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, December 2004.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *Proce. of the second ACM SIGPLAN conference on History of programming languages*, pages 37–52. ACM Press, 1993.

[DC01] Daniel Diaz and Philipe Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 6, October 2001.

[DH05] Daniel Diaz and Rémy Haemmerlé. *GNU Prolog RH user's manual*. INRIA, 1999–2005.

[Dia95] D. Diaz. *Étude de la Compilation des Langages Logiques de Programmation par Contraintes sur les Domaines Finis : le Système clp(FD)*. Thèse de doctorat d'état, Universit ? d'Orl ?ans, 1995.

[Dia03] Daniel Diaz. *GNU Prolog user's manual*, 1999–2003.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 :115–138, 1971.

[DSH90] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1-2) :75–93, 1990.

[Fag96] François Fages. *Programmation Logique par Contraintes*. Collection Cours de l'Ecole Polytechnique. Ed. Ellipses, Paris (192p), 1996.

[Fag00] François Fages. Concurrent constraint programming and linear logic (invited talk). In *Proceedings of ACM Sigplan conference on Principles and Practice of Declarative Programming PPDP'2000*, Montreal, Canada, September 2000. ACM Publishing Company.

[Fag04] François Fages. SiLCC a concurrent constraint programming language based on linear logic. Franco-Japanese Workshop on Constraint Programming, October 2004. Tokyo, Japan.

[FC01] François Fages and Emmanuel Coquery. Typing constraint logic programs. *Journal of Theory and Practice of Logic Programming*, 1(6) :751–777, November 2001.

[FRS01] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming : operational and phase semantics. *Information and Computation*, 165(1) :14–41, February 2001.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3) :95–138, October 1998.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[GMM+78] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in lcf. In *POPL*, pages 119–130, 1978.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Hae05] Rémy Haemmerlé. SiLCC is linear concurrent constraint programming (doctoral consortium). In Maurizio Gabbrielli and Gopal Gupta, editors, *Proceedings of International Conference on Logic Programming ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 448–449. Springer-Verlag, 2005.

[HF06] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[HF07] Rémy Haemmerlé and François Fages. Abstract critical pairs and confluence of arbitrary binary relations. In *Proceedings of th 18th international conference on Rewriting Techniques and Applications, RTA'07*, number 4533 in Lecture Notes in Computer Science, pages 214–228. Springer-Verlag, 2007.

[HFS07] Rémy Haemmerlé, François Fages, and Sylvain Soliman. Closures and modules within linear logic concurrent constraint programming. In *Proceedings of the 27th onference on foundations of software technology and theoretical computer science FSTTCS 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[HJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Marìa M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell : a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5) :1–164, 1992.

[HMS92] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP(R) and some electrical engineering problems. *Journal of Automated Reasoning*, 9(2) :231–260, 1992.

[HMSY87] Nevin Heintze, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap. *CLP(R) Programmer's Manual*. Monash University, Clayton, Victoria, Australia, 1987.

[Hol92] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. Technical Report TR-92-23, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1992.

[Hol95] C. Holzbaur. OEFAI clp(Q,R) manual rev. 1.3.3. Technical Report TR-95-09, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1995.

[HSD98] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3) :139–164, 1998.

[Int95] International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 1 : General core*, 1995. ISO/IEC 13211-1.

[Int00] International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 2 : Modules*, 2000. ISO/IEC 13211-2.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[JMMS98] J. Jaffar, M .Maher, K .Marriott, and P. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3) :1–46, October 1998.

[JMSY92] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3) :339–395, July 1992.

[Kow74] R. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *POPL '93 : Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM Press, 1993.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[Ler07] Xavier Leroy. *The Objective Caml system.* INRIA, 2007.

[LM92] Cosimo Laneve and Ugo Montanari. Mobility in the cc-paradigm. In *Proc. 17th International Symposium on Mathematical Foundations of Computer Science*, volume 629 of *Lecture Notes in Computer Science*, pages 336–345. Springer-Verlag, 1992.

[Mah87] Michael J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of ICLP'87, International Conference on Logic Programming.* MIT Press, 1987.

[MFP97] Kim Marriott Moreno Falaschi, Maurizio Gabbrielli and Catuscia Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2) :281–315, 1997.

[Mil89] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.

[Mil90] Robin Milner. Function as processes. In *Proc. of ICALP*, Springer-Verlag, 1990.

[Mil94] Dale Miller. A proposal for modules in lambda prolog. In *Proceedings of the 1993 Workshop on Extensions to Logic Programming*, volume 798 of *Lecture Notes in Computer Science*, pages 206–221, 1994.

[Mil99] Robin Milner. *Communicating and mobile systems : the pi-calculus.* Cambridge University Press, 1999.

[MMR95] Martin Müller, Tobias Müller, and Peter Van Roy. Multi-paradigm programming in Oz. In *Visions for the Future of Logic Programming : Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 1995.

[Mou] P. Moura. Logtalk. `http://www.logtalk.org`.

[Mou03] Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language.* PhD thesis, Department of Informatics, University of Beira Interior, Portugal, September 2003.

[MP89] Luís Monterio and António Porto. Contextual logic programming. In *Proceedings of ICLP'1989,International Conference on Logic Programming*, pages 284–299, 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1), 1992.

[MTHM97] Robin Milner, Mads Tofte, Rober Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[O'K85] Richard A. O'Keefe. Towards an algebra for constructing logic programs. In *Symposium on Logic Programming*, pages 152–160. IEEE, 1985.

[RF97] Paul Ruet and François Fages. Concurrent constraint programming and mixed non-commutative linear logic. In *Proc. of Computer Science Logic CSL'97*, volume 1414 of *Lecture Notes in Computer Science*, pages 406–423. Springer-Verlag, August 1997.

[Rue97] Paul Ruet. *Logique non-commutative et programmation concurrente par contraintes*. PhD thesis, Université Denis Diderot, Paris 7, 1997.

[Sa03] Konstantinos Sagonas and al. *The XSB System Version 2.5 - Volume 1 : Programmer's Manual*, 1993 – 2003.

[Sar93a] Vijay Saraswat. A brief introduction to linear concurrent constraint programming. Xerox PARC, 1993.

[Sar93b] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[Sch99] Vincent Schächter. *Programmation concurrente avec contraintes fondée sur la logique linéaire*. PhD thesis, Université d'Orsay, Paris 11, 1999.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys.*, 21(3) :413–510, 1989.

[SL92] Vijay A. Saraswat and Patrick Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

[Sol01] Sylvain Soliman. *Programmation concurrente avec contraintes et logique linéaire*. PhD thesis, Université Paris 7, Denis Diderot, April 2001.

[Sol04] Sylvain Soliman. Pi-calcul et LCC, une odyssée de l'espace. In Fred Mesnard, editor, *Programmation en logique avec contraintes (Actes de JFPLC 2004)*, pages 201–218, Angers (France), June 2004. Hermès.

[SRP91] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *POPL'91 : Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991.

[Sti88] Mark E. Stickel. A prolog technology theorem prover : implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 44 :353–380, 1988.

[SW92] D. T. Sannella and L. A. Wallen. a calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, pages 147–177, 1992.

[SW04] Tom Schrijvers and David S. Warren. Constraint handling rules and table execution. In *Proceedings of ICLP'04, International Conference on Logic Programming*, pages 120–136, Saint-Malo, 2004. Springer-Verlag.

[Swe03] Swedish Institute of Computer Science. *Quintus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 2003.

[Swe04] Swedish Institute of Computer Science. *SICStus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 1991–2004.

[Tro92] Anne S. Troelstra. *Lectures on Linear Logic*, volume 29 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, California, 1992.

[Van89] Pascal Van Hentenryck. *Constraint satisfaction in Logic Programming*. MIT Press, 1989.

[VSCA00] Rogério Reis Víctor Santos Costa, Luís Damas and Rúben Azevedo. *YAP user's manual*, 1989–2000.

[War82] David H. D. Warren. Higher-order extensions to Prolog : Are they needed ? In *Machine Intelligence*, volume 10 of *Lecture Notes in Mathematics*, pages 441–454. 1982.

[War83] D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, 1983.

[Wie04] Jan Wielemaker. *SWI Prolog 5.4.1 Reference Manual*, 1990–2004.

Liste des symboles

$\Box$ conjonction d'atomes PLC vide , page 23

$\mathcal{A}_{\dot{c}}(X)$ variables accessibles dans $\dot{c}$ à partir de X , page 112

$\mathcal{A}^1_{\dot{c}}(X)$ variables directement accessibles dans $\dot{c}$ à partir de X , page 112

$\mathcal{A}^s_{\dot{c}}(X)$ variables accessibles par substitution dans $\dot{c}$ à partir de X , page 111

$\mathcal{A}^u_{\dot{c}}(X)$ variables accessibles par unification dans $\dot{c}$ à partir de X , page 111

$\forall \vec{x}(c \rightarrow A)$ *ask* , page 41

$\forall \vec{x}(c \Rightarrow A)$ *ask* persistant , page 41

$\exists$ opérateur de localisation , page 41

$||$ composition parallèle , page 41

$A :\!- c \mid \alpha.$ clause PLC , page 23

$\mathcal{C}, \mathcal{D}$ systèmes de contraintes , page 37

$\langle X; c; \Gamma \rangle$ configuration , page 43

$\equiv$ équivalence structurelle , page 43

$\longrightarrow$ relation de transition LCC , page 43

$\mathcal{M}$ ensemble de contraintes de synchronisation, page 104

$\Vdash_{\mathcal{C}}$ axiomes non-logiques du système de contraintes $\mathcal{C}$, page 37

$\vdash_{\mathcal{C}}$ système d'inférence du système de contraintes $\mathcal{C}$, page 37

$\approx_{\mathcal{D}}$ équivalence observationnelle MLCC vis à vis $\mathcal{D}$, page 129

$\sim_{\mathcal{D}}$ équivalence observationnelle LCC vis-à-vis $\mathcal{D}$, page 78

$\lambda x.M$ λ-abstraction , page 62

MN composition de λ-terme , page 62

L, M, N λ-termes , page 62

U, V, W valeurs du λ-calcul , page 62

ϵ séquence de position vide , page 63

$\mathrm{Pos}(M)$ ensemble des positions du terme M , page 63

$M\mid_p$ sous terme de M à la position p , page 63

$\rightarrow_{\beta_V}$ relation de réduction en « appel par valeur » du λ-calcul , page 63

$\mathcal{O}^{\mathrm{ca}}$ ensemble des contraintes accessibles , page 44

$\mathcal{O}^{\mathcal{D}}$ ensemble des $\mathcal{D}$-contraintes accessibles , page 44

$\mathcal{O}^{D\mathrm{succ}}$ ensemble des $\mathcal{D}$-succès , page 44

$\mathcal{O}^{\mathrm{ps}}$ ensemble des pseudo-succès , page 44

$\mathcal{O}^{\mathrm{succ}}$ ensemble des succès , page 44

$(\nu x)P$ restriction dans le π-calcul , page 76

$\bar{x}z$ envoie de message dans le π-calcul , page 76

$\longrightarrow_{\pi}$ relation de transition du π-calcul , page 76

$P \parallel P$ composition parallèle dans le π-calcul , page 76

$P + P$ choix indéterministe dans le π-calcul , page 76

$x(y)P$ réception de message dans le π-calcul , page 76

Σ_F ensemble de symboles de fonctions , page 37

$\dagger$ traduction logique des agents LCC , page 51

$\diamond$ traduction des contraintes ILL en contraintes classique , page 38

$\star$ traduction des contraintes classique en contraintes ILL , page 38

$\uplus$ union disjointe , page 37

$\mathcal{V}(F)$ ensemble de variables libres dans la formule F , page 37

$\mathcal{V}_{\mathcal{M}}(\dot{c})$ ensemble des variables quantifiables dans la contraintes $\dot{c}$, page 105

$\mathcal{V}$ ensemble de variables , page 37

$\mathcal{V}_e$ variable d'encodage , page 63

$\mathcal{V}_\lambda$ variable du λ-calcul , page 63

$w, x, y, z, \ldots$ variables , page 37

$\vec{x}$ séquence de variables , page 37