ORSAY

N° d'ordre :

UNIVERSITE DE PARIS-SUD
U.F.R. SCIENTIFIQUE D'ORSAY

THESE

présentée

Pour obtenir

Le GRADE de DOCTEUR EN SCIENCES DE L'UNIVERSITE PARIS XI ORSAY

PAR

Laurent PERRON

SUJET : `CC(`$\mathcal{M}$`)`, Un noyau parallèle pour l'implantation des langages de contraintes concurrents

Soutenue le 9 juillet devant la Commission d'examen

MM	Philippe Codognet	Rapporteur
	François Fages	Directeur
	Jean-Louis Imbert	Président
	Andreas Podelski	Rapporteur
	Jean-François Puget	
	Ralf Treinen	

À tous ceux que j'aime et qui me le rendent bien.

Remerciements

Qu'il me soit permis avant tout de remercier de fond du cœur tous ceux qui m'ont aidé dans ce travail. En particulier, j'exprime toute ma gratitude envers mon directeur de thèse François Fages pour sa patience et l'attention soutenue dont il a fait preuve tout au long de ce périple. Je tiens aussi à citer le LIENS pour son accueil et tout ce qu'il a pu m'apporter aussi bien du point de vue scientifique que du point de vue humain, matériel et administratif. Je voudrais enfin remercier les rapporteurs Andreas Podelski et Philippe Codognet qui m'ont fait l'honneur de rapporter mon mémoire. À toutes ces personnes et à toutes celles que je n'ai pas citées, merci.

Résumé

Cette thèse consiste en la description, la définition et l'évaluation d'une implantation *parallèle* des langages de programmation concurrente par contraintes. Cette implantation est basée sur un langage noyau CC($\mathcal{M}$) qui servira de base commune pour l'implantation de différents domaines de contraintes comme les domaines finis, les booléens, l'arithmétique réelle ou d'autres domaines encore. En particulier, il sera présenté à la fin de la thèse une implantation parallèle d'un fragment des domaines finis ainsi qu'une application usuelle de ce domaine de contraintes aux problèmes d'ordonnancements disjonctifs.

Le langage CC($\mathcal{M}$) est le produit de quatre influences distinctes : la programmation logique par contraintes (PLC) [61], les implantations parallèles des langages logiques (PARLOG [29], GHC [110], FGHC [77], Muse [3], les langages concurrents par contraintes (CC) [91, 93] et les travaux sur les langages orientés-objets (Claire [21], SmallTalk [51], Objective-Caml [69]). L'influence de la programmation logique par contraintes est visible dans l'architecture du langage et en particulier au niveau de l'intégration des résolveurs de contraintes dans le langage. Le langage CC($\mathcal{M}$) est défini comme un langage noyau objet parallèle au dessus duquel les domaines de contraintes sont implantés comme des librairies. L'implantation de ce langage repose sur deux paradigmes de programmation : les langages objets et les langages logiques parallèles. C'est grâce aux techniques de ces deux mondes que le langage CC($\mathcal{M}$) a pu être efficacement implanté. D'une part, il bénéficie grâce aux travaux sur le typage des langages objets d'une implantation particulièrement optimisée des structures de données (construites avec des objets). D'autre part, à partir de l'expérience collectée au cours des deux dernières décennies dans l'implantation des langages logiques parallèles et grâce à l'évolution des systèmes d'exploitations et des architectures des ordinateurs, il a été possible de définir une technique originale et compétitive d'implantation du parallélisme-ou basée sur la programmation de la MMU. Enfin, le cadre des langages CC a été développé et utilisé pour donner une sémantique algébrique au langage CC($\mathcal{M}$).

Chapitre I

Introduction

1. La programmation concurrente par contraintes

1.1. Un domaine de recherche en plein essor

Le domaine de mes recherches est la programmation par contraintes. Cette branche de l'informatique issue de la programmation logique connaît depuis une dizaine d'années un succès grandissant, succès justifié par une théorie à la fois simple et expressive, mais aussi par des applications industrielles très significatives.

Si le cadre séquentiel est suffisamment mûr et achevé pour être appliqué à l'industrie, les applications bénéficieraient beaucoup du supplément de puissance, d'expressivité et de réactivité que leur apporteraient la concurrence et une implantation parallèle et distribuée. Mais dans ce domaine très difficile, la théorie est encore insuffisamment développée et beaucoup de travail reste à faire.

C'est donc le point de départ du projet `CC(`$\mathcal{M}$`)` que d'étudier et d'avancer la théorie et les techniques d'implantations dans le domaine de la programmation concurrente par contraintes.

1.2. La Programmation Logique par Contraintes comme généralisation de la Programmation Logique

Historiquement, la Programmation Logique par Contraintes (PLC) [62] est apparue comme une extension de la programmation logique (PL). La classe des langages `CLP(X)` paramétrée par un domaine de contraintes `X` a été introduite comme une généralisation des clauses de Horn

$$B \vdash A_1, \ldots, A_n$$

où B et les A_i sont des prédicats du langage en rajoutant dans cette écriture les contraintes. Ces clauses deviennent alors

$$B \vdash c, A_1, \ldots, A_n$$

où c représente une contrainte du langage de contraintes `X` considéré. Le sens de cette formulation est la suivante, pour que B soit valide, il suffit que les A_i soient valides et que la contrainte c soit satisfaite. Ainsi, une étape de résolution SLD est remplacée par une étape de résolution associée à une satisfiabilité d'une contrainte. La simplicité de ce schéma et le caractère naturel de l'extension par rapport à la programmation logique ont fait la force et la beauté de cette formulation de la programmation par contraintes.

1.3. Deux approches pour l'implantation des contraintes

Il y a communication permanente entre le monde des contraintes et celui des prédicats de la programmation logique. Ce qui a généré deux types d'implantations. Le premier type correspond à une implantation des domaines de contraintes au-dessus du langage de programmation logique (en général PROLOG) comme une librairie. Le deuxième type correspond au cas d'un résolveur de contraintes externe (généralement programmé en C) qui communique avec le langage logique au travers d'une interface pré-définie.

Les deux approches présentent toutes les deux des arguments valables. Si la vision d'un domaine de contraintes comme une librairie du langage offre un temps de développement plus court et une intégration plus souple entre contraintes et calcul des prédicats et une extensibilité des domaines de contraintes et de leurs résolveurs, c'est souvent au détriment d'une certaine efficacité. D'un autre côté, si les résolveurs de contraintes externes offrent une vitesse d'exécution plus élevée, c'est au détriment d'une certaine souplesse d'interface avec le langage logique.

Historiquement, la première intégration des contraintes à un langage logique a été réalisée autour de Prolog III [36], `CLP(R)` [60, 62, 61] et de CHIP [45, 44, 54, 55]. Le système `CLP(R)`, comme CHIP [46] consiste en un module complet implanté en C lié à un interpréteur qui s'occupait de la résolution logique et qui appelait les procédures chargées de l'implantation des primitives du domaine de contraintes des rationnels.

Plus tard, en utilisant la possibilité de geler un but (Delay, Freeze), `CLP(FD)` a été implanté au-dessus de Prolog dans [50, 31]. L'année suivante, est apparue une implantation du domaine des rationnels CLP(q,r) [59], implantation qui sera intégrée à SICStus Prolog [100] comme une librairie.

Mais la vitesse est un facteur limitant et plusieurs systèmes `CLP(FD)` seront par la suite réécrits en C et intégrés à Prolog en utilisant son interface avec le C. Ceci se traduira par un gain en vitesse d'un facteur 5 à 7. Ces exemples illustrent bien les deux approches pour l'intégration des domaines de contraintes dans un langage logique. L'approche librairie est souvent utilisée en premier pour prototyper le module. Puis, quand la vitesse est critique et une fois que l'interface avec le langage déclaratif est stable, les résolveurs de contraintes sont réécrits en C et intégrés à Prolog.

Mais au-delà de ce clivage, cette interface peut prendre différents aspects et cette diversité reflète l'étendue des travaux sur ce sujet. L'approche la plus courante est une approche strictement fonctionnelle qui voit les domaines de contraintes comme un type de donnée abstrait dont les fonctions associées sont intégrées à des algorithmes de recherche génériques (séparation et évaluation, recherche premier échec, etc). Cette approche est utilisée dans des langages comme AKL [64, 13] et Oz [75, 76].

Mais il existe d'autres approches plus complexes et plus intéressantes. On peut citer les travaux qui ont donné `clp(FD)` [33] et `clp(`$\mathcal{B}$`)`[32]. Ces travaux ont montré comment les domaines finis et les booléens peuvent être intégrés efficacement à Prolog en étendant les instructions de la WAM.

1.4. Contraintes et concurrence

Les contraintes et la concurrence[1] sont intrinsèquement liées. En effet, les buts gelés ont été introduits dans Prolog II pour gérer la contrainte de dis-équation sur les termes infinis [35]. Puis, quand la théorie des contraintes a été développée et les domaines de contraintes formalisés [62], ces buts gelés ont été intégrés au monde des contraintes sous la forme de gardes dans ALPS [70] et généralisés dans la classe des langages `CC` (X) [91, 93] par l'opérateur ***ask***. Cette généralisation a accru l'expressivité des langages résultants puisqu'il a été possible de coder les algorithmes de propagation de contraintes de `CLP(FD)` dans `CC(FD)` [113], cela a aussi permis de concevoir la théorie des modèles redondants et coopératifs qui sont au coeur de la thèse de Jean Jourdan [65].

La concurrence ainsi introduite par les buts gelés est une forme restreinte de la concurrence que nous appellerons **concurrence par démons**. Dans cette forme de concurrence, l'exécution du programme est interrompue par des co-routines (démons) déclenchées par l'écriture sur une variable. Ces co-routines assurent la propagation des contraintes dans la base de faits. Il n'y a pas réellement de concurrence d'agents au sens où il n'y a pas plusieurs agents indépendants qui coopèrent mais un agent maître et des co-routines esclaves qui l'interrompent. Néanmoins, ces primitives connaîtront un vif succès. Pour s'en convaincre, il suffit de regarder le succès des opérateurs Freeze [14, 8] et Delay en Prolog et de l'effort poursuivi pour les implanter [116].

À cette forme de concurrence, on peut opposer une **concurrence vraie** qui proposerait un calcul équitable d'agents autonomes. La concurrence vraie peut être implantée de manière parallèle ou pseudo-parallèle. Cette forme de concurrence plus puissante permet aussi d'exprimer des algorithmes plus complexes : propagation locale [81], modèles coopératifs [65]. Elle offre un contrôle plus fin sur les algorithmes de propagation et l'ordre d'exécution des démons[2] et elle permet d'améliorer les stratégies d'exploration d'arbres de recherches[3].

Mais cette concurrence vraie complique l'interface entre les domaines de contraintes et les langages déclaratifs, ce qui argumente en faveur d'une implantation intégrée des domaines de contraintes.

---

[1] La concurrence est une notion formelle. Deux agents concurrents peuvent s'exécuter en même temps et on ne peut spécifier un ordre d'exécution. Le parallélisme est une notion physique qui implique que des processus peuvent s'exécuter en parallèle sur des machines multi-processeurs, ou de manière pseudo-parallèle en temps partagé si le système d'exploitation le permet. La distribution est une notion physique qui indique que le calcul s'effectue sur plusieurs espaces mémoires différents, soit sur des machines multi-processeurs à mémoire non partagée, soit sur des réseaux de stations de travail.

[2] Soit une contrainte à l'ajout de laquelle sont attachées deux familles de démons distinctes. On peut supposer que, alternativement, l'une des deux familles peut décider rapidement si on aboutit à une contradiction. Dans ce cas-là, une exécution équitable entre ces deux familles de démons est rentable.

[3] En effet, avec un calcul d'agents équitable, on est capable d'exprimer aisément la stratégie suivante : soit un point de choix possédant n branches, on peut lancer une évaluation partielle de ces n branches avec n agents en parallèle. A la fin de ces évaluations, les agents se synchronisent et seul les p meilleures branches sont gardées. Cette stratégie bénéficie beaucoup d'une formulation à base de vraie concurrence.

2. Le projet CC($\mathcal{M}$)

2.1. Les besoins d'un langage noyau concurrent

Il apparaît judicieux de développer un langage noyau suffisamment efficace et expressif pour programmer dans le même cadre unifié à la fois les résolveurs spécifiques des domaines de contraintes et les algorithmes génériques de recherche de solutions et d'exploration d'un arbre de choix. De plus, ce langage doit implanter la concurrence vraie et donc proposer un environnement parallèle ou pseudo-parallèle.

C'est donc l'hypothèse de départ du projet CC($\mathcal{M}$) [78, 79] de fournir ce langage noyau pour implanter des environnements de programmation concurrente et par contraintes. Pour satisfaire cette ambition, le langage offre une quadruple fonctionalité d'impérativité, d'implantation modulaire des systèmes de contraintes, de gestion du non-déterminisme et de concurrence vraie basée sur une implantation parallèle.

L'impérativité est une conséquence directe du choix d'un langage noyau. En effet, le langage noyau utilise la représentation des contraintes et non plus les contraintes. Et ce qui était des opérations monotones pour les contraintes devient des affectations impératives pour leurs représentations[4].

La **modularité** fait référence à la paramétrisation des langages de contraintes par rapport à un domaine de contraintes. Un langage noyau doit donc offrir une certaine forme de polymorphisme et de modularité pour utiliser différents domaines de contraintes avec les mêmes algorithmes.

Le **non-déterminisme** est essentiel si on veut appliquer le langage CC($\mathcal{M}$) à des problèmes de résolution de contraintes et d'optimisations. Malheureusement, la conjonction de l'indéterminisme, du non-déterminisme et de la concurrence n'a pas de sémantique simple [72]. Il faudra donc s'attacher à donner une sémantique claire au non-déterminisme dans le langage CC($\mathcal{M}$).

La **concurrence vraie** augmente le pouvoir d'expression du langage. Elle repose principalement sur l'idée d'un calcul équitable d'agents. Celui-ci peut s'implanter aisément avec un calcul parallèle de processus. Dans une telle implantation, l'équité est assurée par la librairie du parallélisme employée (par exemple [87]).

2.2. CC($\mathcal{M}$), un langage objet parallèle et non-déterministe

2.2.1. Le choix d'un langage objet et le besoin de typage

Le choix d'un langage objet comme langage noyau est justifié par la possibilité de surcharger les fonctions, ce qui permet de répondre au besoin de généricité et de modularité. Mais cette faculté a un coût qu'il faut minimiser.

Si l'on regarde l'histoire des langages objets, elle oscille entre les langages dynamiquement typés comme SmallTalk [51] et Objective-C et ceux typés statiquement comme Objective-Caml [69], C++ [107]. On trouve aussi des langages partiellement statiques et partiellement dynamiques (Java [108], Claire [21]). Pour ces langages, le but du compilateur est d'obtenir le meilleur typage possible pour être le plus statique possible [19, 20]. D'un côté, le typage fort, s'il permet de retrouver des performances comparables aux plus rapides

[4] Pour s'en convaincre, il suffit de regarder l'implantation de CLP(FD) au dessus de Prolog [31]. Cette implantation utilise les attributs de variables et les modifie de manière impérative

des langages impératifs (C), bride fortement l'expressivité du langage. D'un autre côté, si le typage dynamique apporte une liberté supplémentaire appréciable pour certaines applications (il permet d'implanter des systèmes extensibles dynamiquement tels qu'on les trouve en représentation des connaissances). Le coût du typage dynamique est non négligeable. Tout est affaire de compromis.

Le langage CC($\mathcal{M}$) présente un compromis original sous la forme de deux langages de types d'expressivités différentes. Ces deux langages implantent un typage statique et un typage dynamique. On peut ainsi optimiser le code avec le typage statique et garder une expressivité suffisante avec le second.

2.2.2. Une implantation parallèle originale du non-déterminisme

Mais un langage objet n'est pas suffisant pour les besoins du projet CC($\mathcal{M}$). Le langage doit supporter le non-déterminisme. En outre, suite à la démocratisation des stations de travail multi-processeurs, le langage CC($\mathcal{M}$) est ciblée sur des architectures multi-processeurs à mémoire partagée. Comme le langage CC($\mathcal{M}$) se veut concurrent et parallèle, et qu'il doit pouvoir implanter des algorithmes de recherche en largeur, il ne peut se satisfaire d'une implantation du non-déterminisme basée sur le « backtracking ».

Pour implanter le langage CC($\mathcal{M}$), nous nous sommes inspirés des très nombreux travaux sur les langages logiques parallèles qui ont été réalisés depuis les années 80. Grâce aux progrès combinés des stations de travail et des systèmes d'exploitation, nous avons pu repenser la classification des techniques d'implantation du non-déterminisme dans un contexte parallèle (parallélisme « ou »). Nous avons conçu une méthode hybride des méthodes traditionnelles (copie de pile et tableau d'adresses [85]). Ce qui nous a permis d'implanter d'une manière inédite et efficace la gestion du non-déterminisme. Cette implantation est basée sur des techniques de très bas niveau de programmation des processeurs annexes à l'unité arithmétique et logique au centre de l'ordinateur, programmation rendue possible grâce aux progrès des systèmes d'exploitation. Ainsi, une partie du travail de gestion des environnements multiples liée à la conjonction du non-déterminisme et du parallélisme a pu être reléguée à la partie matérielle de la station de travail, libérant et accélérant par là même la partie logicielle.

2.3. Une sémantique basée sur les CC

2.3.1. Le choix des CC comme cadre mathématique

Le langage CC($\mathcal{M}$) se veut un langage simple. En particulier, il doit pouvoir être décrit par quelques équations sémantiques. Le cadre des langages CC, de par sa simplicité, nous a paru adéquat pour cette tâche.

En ce sens, nous nous sommes attachés à la description du langage CC($\mathcal{M}$) dans le cadre général des CC. Nous avons décrit le langage des objets et des classes comme un système de contraintes $\mathcal{M}$ acceptable au sens des CC. Nous avons exhibé un codage élégant des objets par les messages qui les constituent et qui les définissent. Cette formalisation décrit aussi parfaitement le passage d'informations entre différents agents basé sur la lecture et l'écriture des champs des objets et non sur l'unification entre variables.

2.3.2. Des extensions nécessaires des CC

Pour décrire de manière unifiée l'impérativité et le non-déterminisme, nous nous sommes intéressés à la notion de contrôle dans les CC. En effet, ces deux notions peuvent se traduire par un contrôle sur les interactions entre les agents. Pour l'exprimer, nous avons tracé l'exécution d'un programme CC grâce à une notion de décoration attachée aux contraintes. À l'aide de cette trace, il nous a été possible de traduire la notion de contrôle par une notion de sélection dans une base de faits finale. La base de fait ainsi triée a été montrée conforme à la sémantique du non-déterminisme et des variables impératives.

3. Une thèse en quatre parties

La thèse se découpe en quatre parties.

La première partie présente une extension décorée des systèmes de contraintes. Par ce formalisme, il est maintenant possible de décorer des contraintes et d'annoter un calcul par des informations supplémentaires. Une première application est donnée. Celle-ci développe une sémantique de trace avec la décoration pour donner une sémantique formelle au non-déterminisme angélique. Cette sémantique est ensuite comparée à la sémantique usuelle.

La deuxième partie utilise de nouveau la théorie de la décoration mais dans un but plus ambitieux. En effet, la décoration associée à une opération de sélection a posteriori sur la base de faits finale permet de donner une sémantique algébrique à la séquentialité et aux variables impératives. On obtient donc une sémantique formelle à une extension non monotone des CC.

La troisième partie présente le langage CC($\mathcal{M}$) et son implantation optimisée. Le langage CC($\mathcal{M}$) est un langage objet parallèle et non-déterministe. Ce langage repose sur deux idées principales : un double langage de type pour optimiser le compromis typage-expressivité et une implantation du parallélisme « ou » basée sur une programmation directe de la MMU.

Enfin, la quatrième et dernière partie présente deux applications du langage CC($\mathcal{M}$) afin de tester ses capacités à implanter des résolveurs concurrents sur des domaines de contraintes particuliers. La première application est une implantation de CC ($\mathcal{B}$) où $\mathcal{B}$ est le domaine de booléens. La deuxième partie montre l'implantation d'un fragment de CC(FD) et son application à un problème d'optimisation combinatoire.

Chapitre II

Préliminaires

1. Notions présupposées

1.1. Théorie de l'ordre

ordre partiel

Un ordre partiel est une relation réflexive, antisymétrique et transitive.

Quand un élément x appartient à un ensemble ordonné $\langle \mathbf{E}, < \rangle$, on notera $\overline{x}$ la fermeture supérieure de x, c'est-à-dire l'ensemble des éléments de $\mathbf{E}$ qui sont supérieurs à x et $\underline{x}$ sa fermeture inférieure, c'est-à-dire l'ensemble des éléments de $\mathbf{E}$ qui sont inférieurs à x.

Treillis

On rappelle quelques résultats de la théorie des treillis issus de [7] et de [39, 40]. En particulier les opérateurs de fermeture et les théorèmes d'itération dans un treillis.

Un treillis est un ordre partiel tel que tout ensemble fini admet un plus petit majorant et un plus grand minorant. Le treillis est dit complet si ce résultat est vrai pour tout ensemble, même infini. Un treillis est dit distributif si les opérations de **min** et de **max** commutent entre elles. Le treillis est dit algébrique si chaque élément est la limite supérieure des éléments finis qui lui sont inférieurs.

Dans un treillis, un opérateur de fermeture est une fonction du treillis dans lui-même qui est extensive

$$\forall x, f(x) > x$$

croissante

$$\forall x, y, (x > y) \Rightarrow (f(x) > f(y))$$

et idempotente

$$\forall x, f(f(x)) = f(x)$$

Il est donc clair que l'image d'un point par un opérateur de fermeture est soit lui-même, soit un point fixe qui lui est supérieur. Donc l'opérateur est strictement déterminé par l'ensemble de ses points fixes. Cette définition a l'avantage d'être facilement manipulable. En effet, l'itération chaotique de deux opérateurs de fermetures correspond à l'opérateur défini par l'intersection des points fixes des deux opérateurs composés. Celui-ci existe toujours car $\top$ est toujours point fixe.

1.2. Définition en logique

Tout au long de cette thèse, nous utiliserons un ensemble fini ou dénombrable de variables $\mathcal{V}$. Les éléments de cet ensemble seront dénotés par des lettres majuscules X et Y.

Nous disposerons aussi d'un ensemble fini ou dénombrable d'attributs $\mathcal{A}$. Les éléments de cet ensemble seront dénotés par des lettres minuscules a et b.

Nous disposons aussi d'un ensemble $\mathcal{B}$ de valeurs simples (1, 2, ..., vrai, faux, ...). Ces valeurs sont typées par des types simples (*entier*, *booléen*, ...).

On considérera que ces types sont des ensembles, ce qui nous permettra d'écrire : faux $\in$ *booléen*, $1 \in$ *entier*...

1.2.1. Langage du premier ordre

Nous supposerons connus les fondements de la logique, à savoir la notion de calcul des prédicats, de résolution SLD, de langage du premier ordre, d'algèbre libre de termes. On pourra trouver une présentation complète dans [47].

1.2.2. Domaine de contraintes

Les contraintes représentent en fait deux choses. D'une part, il y a le langage de contraintes qui est un fragment d'un langage logique du premier ordre et d'autre part il y a son interprétation qui est une correspondance entre une certaine structure mathématique et le langage de contraintes précédemment introduit.

Définition II-1. (Langage de contraintes)
On considère le langage du premier ordre défini par

1. **Un ensemble dénombrable S_F de symboles de constantes et de fonctions.**
2. **Un ensemble dénombrable S_C de symboles de prédicats supposé contenir vrai et $=$.**
3. **Un ensemble infini dénombrable de variables $\mathcal{V}$.**

Une *contrainte atomique* est une proposition atomique de ce langage. On suppose donné un ensemble de *contraintes basiques* qui est un ensemble de formules du premier ordre de ce langage contenant les contraintes atomiques, et fermé par re-nommage des variables. Le *langage des contraintes* est la fermeture par conjonction et quantification existentielle de l'ensemble des contraintes basiques.

Maintenant que l'on a défini le langage des contraintes, on peut définir son interprétation dans une structure mathématique donnée.

Définition II-2. (Interprétation d'un langage de contraintes)
L'interprétation d'un langage de contraintes est supposée fixée par le choix d'une structure mathématique $\mathcal{S} = (\mathcal{D}, E, O, R)$ constituée :

1. **d'un domaine $\mathcal{D}$.**
2. **d'un ensemble $E \subset \mathcal{D}$ d'éléments distingués associés à chaque symbole de constante, noté $[c]$ pour $c \in S_F$ d'arité 0.**
3. **d'un ensemble O d'opérateurs sur $\mathcal{D}$ associés à chaque symbole de fonction en respectant les arités, noté $[f] : \mathcal{D}^n \to \mathcal{D}$ pour tout $f \in S_F$ d'arité n.**
4. **d'un ensemble O de relations sur $\mathcal{D}$ associées à chaque symbole de prédicat en respectant les arités, noté $[p] : \mathcal{D}^n \to \{\text{vrai}, \text{faux}\}$ pour tout $p \in S_C$ d'arité n.**

2. Rappel sur les algèbres cylindriques

Les algèbres cylindriques [52, 53] ont été introduites par Henkin, Monk et Tarski pour donner une sémantique algébrique à la programmation logique. Elles ont par la suite été utilisées par Saraswat [94] pour définir les systèmes de contraintes.

La démarche utilisée est la suivante : pour α un ordinal donné, on définit une structure algébrique qui possède les bonnes opérations pour modéliser la quantification existentielle et l'égalité syntaxique entre deux variables logiques. Puis on exhibe un sous-ensemble de cette classe d'algèbres cylindriques nommée algèbres cylindriques ensemblistes qui correspondent aux suites de longueur α dont les éléments appartiennent à un ensemble donné. Enfin, on établit une correspondance entre cette classe algèbre cylindrique ensembliste et les algèbres libres de termes, algèbres munies de la quantification existentielle et de l'égalité entre termes. Cette démarche est très intéressante puisque le résultat est bien une sémantique algébrique des programmes logiques.

2.1. Définitions

2.1.1. Algèbre cylindrique

Définition II-3. (Algèbre cylindrique de dimension α[52, 53])
Soit α un ordinal, une *algèbre cylindrique de dimension* α est une structure algébrique

$$\mathfrak{U} = \langle \mathbf{A}, +, \cdot, -, 0, 1, c_\kappa, d_{\kappa\lambda} \rangle_{\kappa,\lambda<\alpha}$$

telle que 0, 1, et $d_{\kappa\lambda}$ sont des éléments distingués de A (pour tout $\kappa, \lambda < \alpha$) et telle que +, $\cdots$ sont des opérations binaires sur A et telle que les postulats soient vérifiés pour tout $x, y \in A$ et pour tout $\kappa, \lambda < \alpha$.

(C1) la structure $\langle \mathbf{A}, +, \cdot, -, 0, 1 \rangle$ est une algèbre booléenne.

(C2) $c_\kappa\ 0 = 0$.

(C3) $x \leqslant c_\kappa x$ (c'est-à-dire $x + c_\kappa x = c_\kappa x$).

(C4) $c_\kappa(x \cdot c_\kappa y) = c_\kappa x + c_\kappa y$.

(C5) $c_\kappa c_\lambda x = c_\lambda c_\kappa x$.

(C6) $d_{\kappa\kappa} = 1$.

(C7) si $\kappa \neq \lambda, \mu$ alors $d_{\lambda\mu} = c_\kappa(d_{\lambda\kappa} \cdot d_{\kappa\mu})$.

(C8) si $\kappa \neq \lambda$ alors $c_\kappa(d_{\kappa\lambda} \cdot x) \cdot c_\kappa(d_{\kappa\lambda} \cdot -x) = 0$

Les c_κ sont les opérateurs de cylindrification, les $d_{\lambda\kappa}$ sont les éléments diagonaux.

Cette classe d'algèbre est très importante pour donner une sémantique à la programmation logique et à la programmation par contraintes. Pour cela, on étudiera par la suite une instance particulière de cette structure formelle : les algèbres cylindriques ensemblistes.

2.1.2. Algèbre cylindrique ensembliste

On définit les algèbres cylindriques ensemblistes :

DÉFINITION II-4. (ALGÈBRE CYLINDRIQUE ENSEMBLISTE)

(i) Soit un ensemble $\mathbf{U}$ et un ordinal α. Pour tout $\kappa < \alpha$, on dénote par $\mathbf{C}_\kappa^{\mathbf{U},\alpha}$, ou $\mathbf{C}_\kappa^{(\mathbf{U})}$ ou encore plus simplement $\mathbf{C}_\kappa$ la fonction de $\mathcal{P}({}^\alpha\mathbf{U})$ dans $\mathcal{P}({}^\alpha\mathbf{U})$ telle que

$$\mathbf{C}_\kappa X = \{y \mid y \in {}^\alpha\mathbf{U}, \exists x \in X, \forall \lambda \neq \kappa, x_\lambda = y_\lambda\}$$

(ii) Pour tout ensemble $\mathbf{U}$ et tout ordinal α, pour tout $\lambda, \kappa < \alpha$, on dénote l'ensemble suivant

$$\{y \mid y \in {}^\alpha\mathbf{U}, y_\kappa = y_\lambda\}$$

par $\mathbf{D}_{\kappa\lambda}^{(\mathbf{U},\alpha)}$, ou $\mathbf{D}_{\kappa\lambda}^{(\mathbf{U})}$, ou plus simplement encore $\mathbf{D}_{\kappa\lambda}$.

(iii) $\mathbf{A}$ est un *champ d'ensembles de dimension* α si et seulement si il existe un ensemble $\mathbf{U}$ appelé la base de $\mathbf{A}$, tel que $\mathbf{A}$ soit un sous-ensemble de l'ensemble des suites de longueur α à valeur dans $\mathbf{U}$ (soit $\mathcal{P}({}^\alpha\mathbf{U})$) et tel que $\mathbf{U}$ soit clos pour les opérations d'union, d'intersection, de complémentation, de cylindrification et tel que $\mathbf{U}$ contienne tous les ensembles $\mathbf{D}_{\kappa\lambda}$.

(iv) $\mathfrak{U}$ est une algèbre cylindrique ensembliste de dimension α si et seulement si $\mathfrak{U} = \langle \mathbf{A}, \cup, \cap, \complement, 0, {}^\alpha\mathbf{U}, \mathbf{C}_\kappa, \mathbf{D}_{\kappa\lambda}\rangle$ où $\mathbf{A}$ est un champ d'ensembles de dimension α de base $\mathbf{U}$. Dans le cas où $\mathbf{A} = \mathcal{P}(\mathbf{U})$, on appellera $\mathbf{A}$ et $\mathfrak{U}$ un champ d'ensemble saturé et une algèbre cylindrique ensembliste saturée.

Comme on le voit, les éléments diagonaux des algèbres cylindriques ensemblistes sont des hyperplans de l'ensemble des suites de longueur α à valeur dans l'ensemble de référence, et les opérateurs de cylindrification transforment un ensemble de suite en un cylindre en permettant à une coordonnée donnée de bouger librement (c'est la cylindrification parallèlement à la κ-ième coordonnée). Il reste à vérifier qu'une algèbre cylindrique ensembliste est une algèbre cylindrique.

THÉORÈME II-5.
Toute algèbre cylindrique ensembliste de dimension α est une algèbre cylindrique de dimension α.

Preuve : Voir [52]. □

2.2. Une perspective algébrique des systèmes de contraintes

Mais dans le cas de la programmation par contraintes, cette démarche se révèle inadaptée. En effet, pour modéliser les systèmes de contraintes, on a besoin d'une structure plus générale que les algèbres cylindriques, à savoir les systèmes de contraintes cylindriques [94, 42].

2.2.1. DÉFINITION USUELLE

On résume les définitions présentées dans le chapitre II.

DÉFINITION II-6. (SYSTÈME DE CONTRAINTES CYLINDRIQUE AVEC ÉLÉMENTS DIAGONAUX)
Un système de contraintes cylindrique avec éléments diagonaux est une structure $\langle \mathfrak{C}, \leqslant, \mathcal{V}, \{ \exists_X : \mathfrak{C} \to \mathfrak{C} \}_{X\in\mathcal{V}}, \{ d_{XY} \}_{X,Y\in\mathcal{V}}\rangle$ telle que :

- $\langle \mathfrak{C}, \leqslant \rangle$ est un treillis algébrique complet.
- $\mathcal{V}$ est un ensemble dénombrable de variables.
- d_{XY} sont des éléments distingués de $\mathfrak{C}$.
- $\forall X, Y, Z \in \mathcal{V}, \forall c, d \in \mathfrak{C}$, les axiomes suivants sont vérifiés :
  1. $\exists_X c \leqslant c$.
  2. si $c \leqslant d$ alors $\exists_X c \leqslant \exists_X d$.
  3. $\exists_X(\exists_Y c) = \exists_Y(\exists_X c)$.
  4. $\exists_X(c \sqcup \exists_X d) = \exists_X c \sqcup \exists_X d$.
  5. d_{XX} = vrai.
  6. si $Z \neq X, Y$ alors $d_{XY} = \exists_Z(d_{XZ} \sqcup d_{ZY})$.
  7. si $X \neq Y$ alors $c \leqslant d_{XY} \sqcup \exists_X(c \sqcup d_{XY})$.

REMARQUE II-7. (ALGÈBRES CYLINDRIQUES)
Il y a plusieurs différences entre les algèbres cylindriques. Elles sont issues de la différence entre le discours de la logique et celui des contraintes. Dans le premier discours, les fonctions « et » et « ou » commutent et la négation est définie pour toutes les propositions. Dans le deuxième, il n'en est pas de même. D'une part, le treillis des contraintes muni des opérations de conjonction et de disjonction n'est pas distributif, et d'autre part, la négation n'est pas toujours définie pour un domaine de contraintes donné. Cette affaiblissement s'est donc traduit par un affaiblissement similaire de la définition des systèmes de contraintes cylindriques.

Le plus petit élément de $\mathfrak{C}$ représente vrai, le plus grand faux. Les d_{XY} sont appelés les éléments diagonaux, les $\exists_X$ sont les opérateurs de cylindrification. L'opération $\sqcap$ correspond à la plus grande information commune à deux éléments, et l'opération $\sqcup$ correspond au max de deux éléments, soit la plus grande information que l'on peut déduire de la conjonction de deux éléments.

L'opération $\sqcap$ ne correspond pas exactement au « ou » logique. La notion de *min* ($\sqcap$) de deux éléments est à rapprocher de la disjonction constructive. En clair, $a \sqcap b$ est l'information ou la contrainte que l'on peut déduire du fait que l'on est sûr d'avoir a ou b. Par exemple :

$$(x + y \geqslant 0) \sqcap (x - y \geqslant 0) \vdash (x \geqslant 0)$$

On notera $\vdash$ l'inverse de la relation $\leqslant$.

REMARQUE II-8. (SYSTÈMES D'INFORMATION)
Cette définition est en fait très proche des *systèmes d'informations* tels qu'ils ont été introduits dans l'approche de Dana S. Scott de la théorie des systèmes[97]. Un système de contraintes est un système d'information qui peut devenir inconsistant (ce qui n'est pas possible dans un système d'informations)

2.2.2. LE CAS DES TERMES DE HERBRAND

Le système est celui d'un langage du premier ordre $\mathfrak{L}$ avec égalité. On peut définir plusieurs relations $\vdash$:

Relation 1 : La relation $\vdash$ la plus simple est celle induite par les axiomes de l'égalité dans un langage du premier ordre.

Relation 2 : On peut compliquer la première relation en rajoutant un ensemble d'équations E et en quotientant $\mathcal{D}$ par E. La relation $\vdash$ est en fait la plus petite relation de conséquence logique qui vérifie les axiomes de l'égalité ainsi que E. Le résultat $\langle \mathcal{D}/E, \vdash \rangle$ est bien un système de contraintes.

Il est facile de voir que dans le cas d'un système de contraintes basé sur un langage du premier ordre et contenant un symbole de quantification existentielle, la restriction de cette quantification sur les formules finies vérifie les conditions de la définition précédente. La quantification existentielle est bien une opération de cylindrification.

Dans le cas du système de Herbrand, les formules $X = Y$ où X et Y sont des éléments de $\mathcal{V}$ sont exactement les éléments diagonaux.

Cette remarque donne une intuition sur l'origine de ces formules. La première est évidente. La deuxième indique simplement que l'égalité est symétrique. La troisième est une conséquence de la transitivité de l'égalité, à ceci près que l'on doit penser à protéger la variable qui sert de transition. La troisième montre comment avec deux opérateurs diagonaux, on peut contourner une quantification existentielle. Pour comprendre cette dernière formule, il faut se rendre compte que $\exists_X(\phi \cup \{d_{XY}\})$ correspond en fait à la formule $\phi[X \mapsto Y]$, c'est-à-dire la formule ϕ où toutes les occurrences libres de la variable X sont substituées par la variable Y.

2.2.3. Les systèmes de contraintes cylindriques ensemblistes

Dans cette section, nous allons expliciter une autre connexion entre les systèmes de contraintes cylindriques et la programmation logique par contraintes.

Définition

A l'instar des algèbres cylindriques ensemblistes, on définit les systèmes de contraintes cylindriques ensemblistes.

Définition II-9. (Système de contraintes cylindrique ensembliste [42])

(i) Soit un ensemble $\mathbf{U}$ et un ordinal α. Pour tout $\kappa < \alpha$, on dénote par $\mathbf{C}_\kappa^{\mathbf{U},\alpha}$, ou $\mathbf{C}_\kappa^{(\mathbf{U})}$, ou encore plus simplement $\mathbf{C}_\kappa$, la fonction de $\mathcal{P}({}^\alpha\mathbf{U})$ dans $\mathcal{P}({}^\alpha\mathbf{U})$ telle que

$$\mathbf{C}_\kappa X = \{y \mid y \in {}^\alpha\mathbf{U}, \exists x \in X, \forall \lambda \neq \kappa, x_\lambda = y_\lambda\}$$

(ii) Pour tout ensemble $\mathbf{U}$ et tout ordinal α, pour tout $\lambda, \kappa < \alpha$, on dénote l'ensemble suivant

$$\{y \mid y \in {}^\alpha\mathbf{U}, y_\kappa = y_\lambda\}$$

par $\mathbf{D}_{\kappa\lambda}^{(\mathbf{U},\alpha)}$ ou $\mathbf{D}_{\kappa\lambda}^{(\mathbf{U})}$ ou plus simplement encore $\mathbf{D}_{\kappa\lambda}$.

(iii) $\mathbf{A}$ est un *champ faible d'ensembles de dimension* α si et seulement si il existe un ensemble $\mathbf{U}$ appelé la base de $\mathbf{A}$ tel que $\mathbf{A}$ soit un sous-ensemble de l'ensemble des suites de longueur α à valeur dans $\mathbf{U}$ (soit $\mathcal{P}({}^\alpha\mathbf{U})$), clos pour les opérations d'union, d'intersection, de cylindrification et contenant tous les ensembles $\mathbf{D}_{\kappa\lambda}$.

(iv) $\mathfrak{U}$ est un système de contraintes cylindrique ensembliste de dimension α si et seulement si $\mathfrak{U} = \langle \mathbf{A}, \cup, \cap, \emptyset, {}^{\alpha}\mathbf{U}, \mathbf{C}_\kappa, \mathbf{D}_{\kappa\lambda} \rangle$ où $\mathbf{A}$ est un champ faible d'ensembles de dimension α de base $\mathbf{U}$. Dans le cas où $\mathbf{A} = \mathcal{P}(\mathbf{U})$, on appellera $\mathbf{A}$ et $\mathfrak{U}$ un champ faible d'ensembles saturé et un système de contraintes cylindrique ensembliste saturé.

REMARQUE II-10. (CARACTÈRE FAIBLE D'UN CHAMP D'ENSEMBLES)
Un champ d'ensembles est dit faible s'il n'est pas stable par complémentation.

Le résultat est bien un système de contraintes cylindrique.

Construction canonique d'un système de contraintes cylindrique ensembliste à partir d'un domaine de contraintes

Pour cela, on part d'un langage de contraintes (S_F, S_C, $\mathcal{V}$) muni d'une interprétation dans une structure mathématique fixée $\mathcal{S} = (\mathcal{D}, E, O, R)$. On construit à partir de cette interprétation un système de contraintes cylindrique ensembliste, apte à être utilisé par un `CC` basé sur ce langage de contraintes.

On considère donc le système de contraintes cylindrique ensembliste saturé à base $\mathcal{D}$. Cet ensemble de suites est un support idéal pour définir les contraintes. Une contrainte c est un sous-ensemble de cet ensemble de suites. On définit les variables libres de cette contrainte c comme les ordinaux λ tels que $\mathbf{C}_\lambda c \neq c$.

Maintenant, soit d une contrainte du langage de contraintes d'arité p. Cette contrainte possède dans l'interprétation mathématique du langage de contraintes une table de vérité, c'est-à-dire un ensemble H de p-uples qui satisfait d. A partir de là, soit $\lambda_1, \ldots, \lambda_n$ des variables vues comme positions dans les suites de longueur ω, on peut définir l'image de $c(\lambda_1, \ldots, \lambda_p)$ comme étant l'ensemble des suites qui se projettent sur un p-uple de H. C'est à dire

$$[[c(\lambda_1, \ldots, \lambda_p)]] = \{y \in {}^{\alpha}\mathcal{D} \mid \exists (x_1, \ldots, x_p) \in H, \forall i \in [1..p], y_{\lambda_i} = x_i\}$$

Cette définition est une adaptation de la consistance d'arc appliquée point par point sur le système de valeur des variables de la contrainte. Mais le caractère même de son application point par point traduit l'adéquation entre l'interprétation du domaine de contraintes et les systèmes de contraintes ainsi générés.

Appliquer une contrainte c à une base de faits (c'est-à-dire un sous ensemble de ${}^{\alpha}\mathcal{D}$) revient donc à considérer l'intersection entre la base de faits et l'interprétation de la contrainte. La conjonction de deux contraintes est l'intersection de leur interprétation. La disjonction est égale à leur union.

Cette présentation simplifie beaucoup la présentation des contraintes. En effet, il devient facile de définir la conséquence logique et l'équivalence de contraintes. La contrainte c implique la contrainte d si et seulement si $[[c]] \subseteq [[d]]$. La contrainte c est équivalente à la contrainte d si et seulement si $[[c]] = [[d]]$.

REMARQUE II-11. (STRUCTURE DES DOMAINES DE CONTRAINTES)
La construction canonique d'un système de contraintes à partir d'un domaine de contraintes muni de son interprétation rend bien compte de l'idée que pour passer d'un domaine de contraintes à un système de contraintes, il faut finalement s'abstraire des valeurs et ne plus s'intéresser qu'à la structure et aux relations qu'ont les contraintes entre

elles. Cette structure est traduite par la relation d'inclusion entre les sous-ensembles de ${}^{\alpha}\mathcal{D}$.

3. Les langages de contraintes concurrents déterministes

Maintenant que sont définis les systèmes de contraintes, nous allons pouvoir présenter les langages concurrents de contraintes déterministes. Pour cela, nous allons d'abord donner la syntaxe de ces langages pour après en donner une sémantique opérationnelle, ainsi qu'une sémantique algébrique.

3.1. Définition des langages concurrents de contraintes

3.1.1. Syntaxe du langage

La syntaxe du langage concurrent de contraintes est la suivante :

CC déterministe :

Programme	: :=	Déclaration . Agent	*Déclaration et agent initial*
Déclaration	: :=	Déclaration . Déclaration	*Liste de déclarations*
	|	p : : Agent	*Déclaration d'une fonction*
	|	ϵ	*Déclaration vide*
Agent	: :=	***tell***(c)	*Ajout d'une contrainte*
	|	***ask***(c) $\rightarrow$ Agent	*Demande d'une contrainte*
	|	Agent $\parallel$ Agent	*Composition parallèle*
	|	$\exists_X$ Agent	*Variable locale*
	|	p(X)	*Appel fonctionnel*
	|	stop	*Agent nul*

Par la suite on notera plus simplement ***tell***(c) par $c\downarrow$ et ***ask***$(c) \rightarrow$ A par $c \rightarrow$ A. Nous allons étudier par la suite la signification de chacun de ces opérateurs.

3.2. Sémantique opérationnelle

On présente la sémantique opérationnelle des constructeurs du langage CC déterministe par un système de transition :

$$\langle \Gamma, \longrightarrow \rangle$$

dont les configurations sont des couples Γ = Agent $\times$ $\mathcal{S}$ (la base de faits courante). Chaque règle de transition sera donnée dans un style SOS [82]. Et chaque opérateur du langage CC déterministe sera décrit par un certain nombre de transitions.

3.2.1. L'opérateur *tell*

On commence par l'opérateur le plus simple ***tell*** c qui ajoute la contrainte c à la base de faits courante. Il convient donc de vérifier si cette base de faits est en mesure de l'accepter, c'est-à-dire si le résultat (base de faits) $\cup c$ n'est pas inconsistant (une base de faits est inconsistante si elle contient la valeur faux). On en déduit les deux règles suivantes :

$$\frac{\textbf{faux} \notin \mathcal{S} \sqcup c}{\langle \textit{tell}\ c, \mathcal{S}\rangle \longrightarrow \langle \text{stop}, \mathcal{S} \sqcup c\rangle}$$

et on dénotera la configuration $\langle \text{stop}, \textbf{faux}\ \rangle$ par **fail**. On s'intéressera plus tard à la manière dont réagissent les autres constructeurs en présence de ce même **fail**.

3.2.2. L'opérateur *ask*

L'opérateur ($\textit{ask}\ c \rightarrow$ Agent) attend que c soit une conséquence de la base de faits courante pour se réduire vers l'agent. Mais si la base de faits implique la négation de la contrainte c (si le langage de contraintes supporte cette notion de négation), alors l'opérateur *ask* peut se réduire en l'agent stop. On en déduit les règles suivantes :

$$\frac{\mathcal{S} \vdash c}{\langle \textit{ask}\ c \rightarrow \text{A}, \mathcal{S}\rangle \longrightarrow \langle \text{A}, \mathcal{S}\rangle}$$

$$\frac{\mathcal{S} \sqcup c \vdash \textbf{faux}}{\langle \textit{ask}\ c \rightarrow \text{A}, \mathcal{S}\rangle \longrightarrow \langle \text{stop}, \mathcal{S}\rangle}$$

3.2.3. Composition parallèle ||

La composition parallèle correspond en fait à une conjonction dont on ne spécifie pas l'ordre d'évaluation. En conséquence, on a les deux règles de transition suivantes pour l'entrelacement à gauche et à droite d'agents en parallèle :

$$\frac{\langle \text{A}, \mathcal{S}\rangle \longrightarrow \langle \text{A'}, \mathcal{S}'\rangle}{\langle \text{A}\|\text{B}, \mathcal{S}\rangle \longrightarrow \langle \text{A'}\|\text{B}, \mathcal{S}'}$$

$$\frac{\langle \text{A}, \mathcal{S}\rangle \longrightarrow \langle \text{A'}, \mathcal{S}'\rangle}{\langle \text{B}\|\text{A}, \mathcal{S}\rangle \longrightarrow \langle \text{B}\|\text{A'}, \mathcal{S}'}$$

Maintenant il faut traiter le cas où un agent en parallèle échoue (c'est-à-dire donne *fail*). Dans ce cas, c'est toute la composition parallèle qui échoue.

$$\frac{\langle \text{A}, \mathcal{S}\rangle \longrightarrow \textit{fail}}{\langle \text{A}\|\text{B}, \mathcal{S}\rangle \longrightarrow \textit{fail}}$$

$$\frac{\langle \text{A}, \mathcal{S}\rangle \longrightarrow \textit{fail}}{\langle \text{B}\|\text{A}, \mathcal{S}\rangle \longrightarrow \textit{fail}}$$

Ce qui traduit le *fail* est un élément adsorbant (A $\|$ *fail* = *fail*) pour l'opération $\|$. Dans le même registre, l'agent stop est un élément neutre de cette même relation (A $\|$ stop = A).

3.2.4. Variable locale $\exists_X$

On peut maintenant rajouter deux règles pour traiter le cas de la cylindrification ($\exists_X$). Pour comprendre ces deux règles, il faut voir la création d'une variable locale comme une membrane entre un agent et le reste du monde, dans cette optique ces deux règles transportent la membrane du côté de l'agent ou du côté du monde.

$$\frac{\langle \mathrm{A}, \exists_X c\rangle \longrightarrow \langle \mathrm{A'}, (\exists_X c) \sqcup d\rangle}{\langle \exists X\, \mathrm{A}, c\rangle \longrightarrow \langle \exists X (d, \mathrm{A'}), c \sqcup \exists_X d\rangle}$$

et sa réciproque

$$\frac{\langle \mathrm{A}, (\exists_X c) \sqcup d\rangle \longrightarrow \langle \mathrm{A'}, (\exists_X c) \sqcup d'\rangle}{\langle \exists X (d, \mathrm{A}), c \sqcup \exists_X d\rangle \longrightarrow \langle \exists X (d, \mathrm{A'}), c \sqcup \exists_X d'\rangle}$$

$\exists X(\mathrm{A}, d)$ représente l'agent A dans une base de faits locale contenant la contrainte d et protégée par une quantification existentielle sur X. Ce qui permet à l'agent à l'intérieur de la base de faits locale de voir pleinement la contrainte qu'il a posée. En particulier, le programme suivant :

$$\exists_X (\textbf{\textit{tell}}(c(X)) \| \textbf{\textit{ask}}(c(X)) \rightarrow \mathrm{B})$$

exécute bien l'agent B car le premier $\textbf{\textit{tell}}(c(X))$ est bien visible par l'$\textbf{\textit{ask}}$ même si sa portée est limitée par la quantification existentielle. La notion de base de faits locale est en fait une notion d'environnement équivalente à la notion de variable locale. C'est exactement le but cherché.

3.2.5. Appel de fonction $p(X)$

$$\frac{\mathrm{p}(Y) :: \mathrm{A}}{\langle \mathrm{p}(X), \mathcal{S}\rangle \longrightarrow \langle \exists_Y (\textbf{\textit{tell}}(d_{XY}) \| \mathrm{A}), \mathcal{S}\rangle}$$

pour l'appel fonctionnel. Ces règles encodent le passage de paramètres lors de l'appel des fonctions. En particulier, il sera courant de choisir lors de la définition des fonctions des variables telles que la première règle ne soit jamais appliquée. Dans ce cas, il y a toujours passage de paramètres lors d'un appel de fonction.

Cette règle s'étend à un nombre quelconque de paramètres, en vérifiant quand même que le nombre de paramètres passé est égal au nombre de paramètres demandé par la fonction.

3.2.6. Agent stop

Il ne reste plus que l'agent stop qui a un comportement assez évident. Il est un élément neutre dans la composition parallèle

$$\frac{\langle \mathrm{A} \| \mathrm{stop}, \mathcal{S}\rangle}{\langle \mathrm{A}, \mathcal{S}\rangle}$$

$$\frac{\langle \mathrm{stop} \| \mathrm{A}, \mathcal{S}\rangle}{\langle \mathrm{A}, \mathcal{S}\rangle}$$

Et s'il ne reste plus qu'un seul agent stop, alors le programme est un succès et la base de faits associée est la base de faits finale ou base de faits calculée.

3.3. Sémantique dénotationnelle des langages concurrents de contraintes

On rappelle ici de manière très informelle comment on peut donner une sémantique dénotationnelle aux programmes CC déterministes. Pour plus de détails, il suffit de relire [94].

Pour définir cette sémantique, on va s'appuyer sur les propriétés des treillis et des opérateurs de fermeture. En effet, dans un treillis, un opérateur de fermeture est déterminé de manière unique par l'ensemble de ses point fixes.

Or dans le cas des CC déterministes, chaque agent peut être vu comme un opérateur de fermeture dans un treillis d'informations. Et cette vision est compositionnelle. C'est-à-dire que le comportement d'un tout est la composition du comportement de ses parties.

Il ne reste plus maintenant qu'à donner la traduction de chaque constructeur des CC sous la forme d'un opérateur de fermeture. Pour cela, on donne les équations sémantiques suivantes qui associent à chaque agent l'ensemble de ses points fixes grâce aux opérations $\mathcal{A}$ sur les agents, $\mathcal{D}$ sur les déclarations et $\mathcal{P}$ sur les programmes pour un environnement e et un ensemble de contraintes $\mathcal{D}$.

Sémantique dénotationnelle des CC déterministes :

$$
\begin{aligned}
\mathcal{A}(c)e &= \{d \in \mathcal{D} \mid d \geqslant c\} \\
\mathcal{A}(\textbf{\textit{ask}}(c) \rightarrow A)e &= \{d \in \mathcal{D} \mid d \geqslant c \Rightarrow d \in \mathcal{A}(A)e\} \\
\mathcal{A}(A \| B)e &= \{d \in \mathcal{D} \mid d \in \mathcal{A}(A)e \wedge d \in \mathcal{A}(B)e\} \\
\mathcal{A}(\exists X A)e &= \{d \in \mathcal{D} \mid \exists c \in \mathcal{A}(A)e, \exists_X d = \exists_X c\} \\
\mathcal{A}(p(X))e &= \exists_\alpha(d_{\alpha X} \sqcup e(p)) \\
\mathcal{D}(\epsilon)e &= e \\
\mathcal{D}(p(X) :: A.D)e &= \mathcal{D}(D)e[p \mapsto \exists_X(d_{\alpha X} \sqcup \alpha(A)e)] \\
\mathcal{P}(D.A) &= \mathcal{A}(A)(\mathcal{D}(D))
\end{aligned}
$$

Chapitre III

Systèmes de contraintes décorés

Annoter les variables d'un calcul par des informations diverses est une idée répandue en programmation. Elle permet de compléter le calcul en cours par un autre calcul qui s'effectue en plus. Dans le cadre de la programmation logique, cette idée a déjà été utilisée et a trouvé plusieurs applications. Elle sert à étendre le mécanisme d'unification [58], à optimiser la gestion mémoire [68] ou à implanter les co-routines [14]. On la retrouve dans le cas de la programmation logique sur machines multi-processeurs pour garder une trace de la localisation (en termes de processeurs) des variables du calcul [57]. Cette idée peut ainsi coder toute sorte d'informations, aussi complexe soient-elles. Par exemple, les attributs de variables sont la base de l'implantation des domaines finis dans SICStus Prolog V2.6 [100] ou dans CLP(FD) [50, 31, 47].

L'idée d'appliquer cette idée à la programmation par contraintes en général et aux cc de Saraswat [93] est donc naturelle. En outre, les langages cc introduisent une abstraction de la CLP dans le sens où ils ne s'intéressent qu'à la structure de l'implication logique. Nous mettons à profit cette abstraction afin d'y ajouter la notion de décoration qui trouve là une sémantique simple : pour annoter un système de contraintes, il suffit de considérer son produit cartésien avec un autre système de contraintes appelé système de décoration. La notion de produit cartésien de système de contraintes cylindriques n'étant pas définie, nous la construirons grâce à deux extensions des systèmes de contraintes cylindriques, les systèmes de contraintes typés et le produit cartésien de systèmes de contraintes..

Cette technique est utile pour tracer l'exécution d'un programme cc. En particulier, il est possible de définir un système de décoration qui traduit exactement la position d'un agent dans un arbre de choix. Ce système de décoration nous permet de coder le non-déterminisme angélique [63] comme un cas particulier d'exécution d'un programme cc déterministe décoré.

1. Extension des systèmes de contraintes

Dans cette section, nous allons étudier certaines extensions que l'on peut adjoindre aux systèmes de contraintes cylindriques, afin d'augmenter les constructions possibles et le pouvoir d'expression des langages de contraintes. Tout d'abord, nous allons fournir aux systèmes de

contraintes cylindriques une notion de typage. Enfin, à partir de cette notion de typage, nous pourrons définir une notion de produit cartésien de systèmes de contraintes.

1.1. Système de contraintes cylindrique typé

L'idée du typage repose en fait sur une triple idée : l'information supplémentaire liée au type qui nous renseigne, la notion de restriction qui nous dit que seulement certaines expressions peuvent être typées, et enfin la notion de vérification qui nous dit que les expressions bien typées sont les expressions correctes ou valides. Ces trois idées sont liées respectivement aux concepts d'expressivité, de vérification de type (et de complétude) et de correction.

1.1.1. Définition du typage

Pour parler de typage, il faut parler de langage de type. Ici nous nous restreignons à un langage de type fini. Cela suffit pour décrire les ensembles typés simplement en logique, ou les ensembles à sortes ordonnées qui sont finis.

Définition III-1. (Système de contraintes cylindrique typé)
Soit un système de contraintes cylindrique

$$\mathfrak{U} = \langle \mathbf{A}, \leqslant, \exists_X, d_{XY} \rangle_{X,Y \in \mathcal{V}},$$

et un langage de type qui est un inf-demi treillis fini

$$\langle \mathfrak{T}, \prec, \top, \bot, \curlywedge \rangle.$$

On peut définir les contraintes de type qui sont des éléments de $\mathfrak{U}$ suivant $\tau_X(t)$ pour $t \in \mathfrak{T}$ et $X \in \mathcal{V}$. Ces éléments doivent vérifier les conditions suivantes :

- $\tau_X(t) \cdot \tau_X(t') = \tau_X(t \curlywedge t')$.
- $\exists_X(\tau_X(t)) =$ **vrai si** $t \neq \bot$.
- $\tau_X(\top) =$ **vrai pour tout** $X \in \mathcal{V}$.
- $\tau_X(\bot) =$ **faux pour tout** $X \in \mathcal{V}$.
- $\tau_X(t) \leqslant \tau_X(t') \Leftrightarrow t' \prec t$.

On suppose aussi que le typage est décidable, c'est-à-dire que l'on peut décider si un élément x de $\mathfrak{U}$ est compatible avec une famille de contraintes de type.

Ces contraintes de type vérifient quelques propriétés :

Prop. 1 : $\Delta(\tau_X(t)) = \{X\}$ si $\bot \prec t \prec \top$[1].

Prop. 2 : $\tau_X(t) \sqcup \delta_{XY} \vdash \tau_Y(t)$.

Prop. 3 : Si $t' \prec t$, $x \sqcup \tau_X(t) \leqslant x \sqcup \tau_X(t')$.

[1] $X \in \Delta(c) \Rightarrow X$ est une variable libre de c

1.1.2. Interprétation ensembliste

Soit $\mathfrak{U}$ un système de contraintes cylindrique ensembliste de base U, et soit H une famille de sous-ensembles de U contenant U et $\emptyset$ close par intersection, on peut définir les contraintes de type $T_\lambda(h)$ pour $h \in H$ et $\lambda < \alpha$ par

$$T_\lambda(h) = \{y \in {}^{\alpha}U \mid y_\lambda \in h\}$$

$$T_\lambda(\top) = {}^{\alpha}U$$

$$T_\lambda(\bot) = \emptyset$$

Lemme III-2.
La famille des $T_\lambda(h)$ ainsi définie est bien une famille de contraintes de type.

En particulier $h \curlywedge h' = h \cap h'$.

1.1.3. Système de contraintes pleinement typé

Soit maintenant un système de contraintes construit avec un ensemble $\mathcal{V}$ de variables. Soit c une contrainte ayant $X_1, \ldots, X_p$ comme variables libres. Et soit $\mathfrak{T}$ un langage de type. On dénommera contrainte totalement typée la contrainte $c \wedge \tau_{X_1}(t_1) \wedge \ldots \wedge \tau_{X_p}(t_p)$, où les t_i sont des éléments de $\mathfrak{T}$ différents de $\top$ et $\bot$. Une base de faits est pleinement typée si elle ne contient que des contraintes pleinement typées.

Les contraintes pleinement typées n'ont donc pas de variables non typées. Cette définition nous servira par la suite pour caractériser les bases de faits qui sont assimilables à des produits cartésiens de systèmes de contraintes.

1.2. Produit cartésien de systèmes de contraintes cylindriques

La difficulté du produit cartésien de systèmes de contraintes vient des variables, des opérateurs diagonaux, et de la quantification existentielle, qui sont tous les trois globaux. En effet, les opérateurs diagonaux doivent lier toutes les variables du produit cartésien, mais ces mêmes variables ne doivent pas être partagées entre des contraintes issues de systèmes de contraintes différents. Une première approche basée sur une union disjointe des variables de chaque système de contraintes du produit est donc inévitablement vouée à l'échec puisque l'on ne pourra pas définir tous les opérateurs diagonaux. Il faut donc un ensemble commun de variables. Mais il est nécessaire d'assurer une certaine indépendance des différentes composantes du produit cartésien. C'est dans cette optique que nous sert le typage des systèmes de contraintes cylindriques. En typant différemment les variables et les formules appartenant à des composantes différentes, on est capable de garder leur globalité tout en s'assurant de la cohérence du produit cartésien.

Soit $(\mathfrak{C}_i, \leqslant_i, \mathcal{V}_i)_{i \in [1..n]}$ une famille finie de systèmes de contraintes cylindriques. On cherche à définir le produit cartésien de cette famille.

1.2.1. Les variables du produit cartésien

On suppose que tous les $\mathcal{V}_i$ sont des ensembles infinis dénombrables. Soit $\mathcal{V}$ un ensemble infini dénombrable de variables. Comme il est de même cardinalité que chacun des $\mathcal{V}_i$, on peut définir une famille de bijection σ_i de $\mathcal{V}_i$ vers $\mathcal{V}$. Ces bijections sont une forme de re-nommage de variables.

Puis à chaque contrainte c de $\mathfrak{C}_i$ on peut associer son écriture renommée par σ_i avec des variables de $\mathcal{V}$. On notera $c\sigma_i$ la contrainte ainsi renommée. D'autre part, si la contrainte c n'a pas de variables libres, on notera son re-nommage (nul) indifféremment c ou $c\sigma_i$.On notera aussi $\Delta(c\sigma_i)$ l'ensemble des variables libres de la contrainte c ainsi renommée.

1.2.2. Définition du produit cartésien saturé

Soit $(\sigma_i)_{i\in[1..n]}$ une famille de fonctions de re-nommage des $\mathcal{V}_i$ sur $\mathcal{V}$. On considère l'ensemble $\mathfrak{C}$ des formules écrites avec les n-uples $(c_1\sigma_1, \ldots, c_n\sigma_n)$, où les c_j sont des éléments de $\mathfrak{C}_j$, et avec les deux opérations associatives commutatives $\sqcup$ et $\sqcap$.

De plus, on normalise chaque élément de $\mathfrak{C}$ en considérant la distributivité de $\sqcup$ par rapport à $\sqcap$

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$

Tout élément s'écrit donc en forme normale disjonctive :

$$(a_{11} \sqcup \ldots \sqcup a_{1i_1}) \sqcap \ldots \sqcap (a_{n1} \sqcup \ldots \sqcup a_{ni_n})$$

Enfin, on munit $\mathfrak{C}$ des lois et des éléments suivants :

Correspondance entre le produit cartésien et les composantes

On peut remarquer les deux règles suivantes qui établissent la correspondance entre les opérations $\sqcup$ et $\sqcap$ des composantes du produit cartésien et celles du système produit :

$$(x_1, \ldots, x_i \sqcup x'_i, \ldots, x_n) = (x_i, \ldots, x_i, \ldots x_n) \sqcup (x_1, \ldots, x'_i, \ldots x_n)$$

$$(x_1, \ldots, x_i \sqcap x'_i, \ldots, x_n) = (x_1, \ldots, x_i, \ldots, x_n) \sqcap (x_1, \ldots, x'_i, \ldots, x_n)$$

Ces deux inégalités traduisent bien la perte d'information qui a lieu quand on passe d'une composante au produit. Car de ces égalités, on peut déduire que

$$(a \sqcup b, c \sqcup d) = (a, c) \sqcup (b, c) \sqcup (a, d) \sqcup (b, d) \neq (a, c) \sqcup (b, d)$$

Ce point sera illustré de manière graphique dans la section 1.3..

Un ordre partiel

On peut associer à $\mathfrak{C}$ un ordre partiel en définissant tout d'abord un ordre sur les n-uples du produit cartésien :

$$(c_1\sigma_1, \ldots, c_n\sigma_n) \leqslant (d_1\sigma_1, \ldots, d_n\sigma_n) \Leftrightarrow c_1 \leqslant_1 d_1 \wedge \ldots \wedge c_n \leqslant_n d_n$$

Puis on peut étendre cet ordre à tous les éléments du $\mathfrak{C}$ en considérant les lois suivantes :

$$a \sqcup b \leqslant c \Longleftrightarrow (a \leqslant c) \wedge (b \leqslant c)$$

$$a \leqslant b \sqcap c \Longleftrightarrow (a \leqslant b) \wedge (b \leqslant c)$$

Les deux règles duales sont à tempérer par la correspondance entre le produit cartésien et ses composantes (paragraphe précédent). On peut déjà poser :

$$a \leqslant b \sqcup c \Longleftarrow (a \leqslant c) \vee (b \leqslant c)$$

$$a \sqcap b \leqslant c \Longleftarrow (a \leqslant c) \vee (b \leqslant c)$$

Les réciproques ne sont pas toujours vraies, ce qui se verra aussi de manière graphique dans la section 1.3..

Un plus grand élément

On définit le plus grand élément de $U = (\top_1, \ldots, \top_n)$.

Un plus petit élément

On définit le plus petit élément de $U = (\bot_1, \ldots, \bot_n)$.

La quantification existentielle

La quantification existentielle se définit par rapport à la quantification existentielle de chaque composante du produit cartésien. Cette opération doit être définie de manière globale. Pour cela, il faut s'assurer que toutes les instances renommées $\sigma_i(X)$ de la variable X quantifiée sont égales. On le fera on forçant l'égalité avec une autre variable.

Ainsi définit-on l'opérateur $\exists_X$ en choisissant une variable X' qui n'apparaît libre dans aucune des c_i suivantes :

$$\exists_X(c_1\sigma_1, \ldots, c_n\sigma_n) = \exists_{X'} \left(\exists_{\sigma_1^{-1}(X)}(c_1 \sqcup d_{\sigma_1^{-1}(X)\sigma_1^{-1}(X')}), \ldots, \exists_{\sigma_n^{-1}(X)}(c_n \sqcup d_{\sigma_n^{-1}(X)\sigma_n^{-1}(X')}) \right)$$

Ce qui avec une variable X' convenablement choisie (c'est-à-dire n'apparaissant jamais dans le calcul), devient équivalent à :

$$\exists_{\sigma_1^{-1}(X)}(c_1 \sqcup d_{\sigma_1^{-1}(X)\sigma_1^{-1}(X')}), \ldots, \exists_{\sigma_n^{-1}(X)}(c_n \sqcup d_{\sigma_n^{-1}(X)\sigma_n^{-1}(X')})$$

Des éléments diagonaux

De même, on définit les éléments diagonaux en posant :

$$d_{XY} = ((d_{\sigma_1^{-1}(X)\sigma_1^{-1}(Y)})\sigma_1, \ldots, (d_{\sigma_n^{-1}(X)\sigma_n^{-1}(Y)})\sigma_n)$$

Il est bien entendu que dans cette formule, toutes les instances des $\sigma_i^{-1}(X)$ ont la même valeur X.

1.2.3. Propriétés du produit cartésien saturé

On a le théorème suivant sur $\mathfrak{C}$:

Théorème III-3. (Produit cartésien de systèmes de contraintes cylindriques) Avec les notations précédentes,

$$\langle \mathfrak{C}, \leqslant, \mathcal{V}, \{\exists_X\}, \{d_{XY}\}\rangle_{X,Y\in\mathcal{V}}$$

est un système de contraintes cylindriques.

Preuve : Il y a plusieurs points à vérifier :

(i) $\langle \mathfrak{C}, \leqslant\rangle$ est un treillis complet algébrique.

- $\langle \mathfrak{C}, \leqslant\rangle$ est un ordre partiel avec les opérations $\sqcap$ et $\sqcup$ ainsi qu'un plus grand élément et un plus petit élément. C'est donc un treillis.
- C'est un treillis algébrique et complet puisque chaque composante est algébrique et complète et que le produit cartésien est de dimension finie.

(ii) Les opérateurs de quantifications existentielles vérifient les propriétés suivantes :

- $\exists_X c \leqslant c$ est vrai car $\forall i \in [1..n], \exists_{\sigma_i^{-1}X} c_i \leqslant_i c_i$.
- si $c \leqslant d$ alors $\exists_X c \leqslant \exists_X d$. Vrai car si $c_i \leqslant_i d_1$, $\exists_{\sigma_i^{-1}X} c_i \leqslant_i \exists_{\sigma_i^{-1}X} d_i$.
- $\exists_X \exists_Y c = \exists_Y \exists_X c$, toujours vrai composante par composante.
- $\exists_X (c \sqcup \exists_X d) = \exists_X c \sqcup \exists_X d$.

(iii) Les éléments diagonaux vérifient les conditions suivantes :

- $d_{XX} = \mathbf{vrai}$. Comme $\mathbf{vrai} = \{\bot_1, \ldots, \bot_n\}$. Il faut montrer que $d_{\sigma_i^{-1}(X)\sigma_i^{-1}(X)} = \bot_i$ pour tout i, ce qui est vrai puisque $\mathfrak{C}_i$ est un système de contraintes cylindrique.
- $d_{XY} = \exists_Z (d_{XZ} \sqcup d_{ZY})$. Cette égalité est prouvée vraie si $Z \neq X, Y$ en utilisant la définition des systèmes de contraintes cylindriques pour chaque composante du produit cartésien.
- De même, on montre la dernière égalité : $X \neq Y$ alors $c \leqslant d_{XY} \sqcup \exists_X (c \sqcup d_{XY})$.

Ainsi donc le produit cartésien saturé de systèmes de contraintes cylindriques est un système de contraintes cylindrique.

□

1.2.4. Fragment pleinement typé du produit cartésien saturé

Comme on a pu le voir, il est possible de construire le produit cartésien $\mathfrak{C}$ d'une famille de systèmes de contraintes cylindriques. Mais ce produit cartésien partage les variables entre les différentes composantes. Or cette construction n'a pas de sémantique claire. En effet, comme les variables sont communes, on peut se demander quel est le sens d'une variable partagée entre deux domaines de contraintes disjoints. Quel est son domaine ? Comment décrire dans ces conditions le lien entre la sémantique de chaque composante et celle du produit cartésien. Il convient donc de « recloisonner » les variables autour de chaque système de contraintes. Pour cela nous allons utiliser les propriétés de typage précédemment introduites.

On se propose donc d'instaurer un cloisonnement strict qui interdit toute communication entre des variables attachées à des domaines de contraintes différents. Pour cela, nous commençons par introduire le langage de type $\mathfrak{T}$ suivant :

$$\mathfrak{T}^n = \{\mathbf{dom}_1, \ldots, \mathbf{dom}_n, \top, \bot\}$$

où tous les $\mathbf{dom}_i$ sont incomparables entre eux. On rajoute donc les contraintes de type à $\mathfrak{C}$. Le résultat obtenu sera dénoté $\mathfrak{C}*$.

On va maintenant pour chaque i et pour chaque contrainte c_i de $\mathfrak{C}_i$ associer son typage $\tau_i(c) = \tau_{\lambda_1}(\mathbf{dom}_i) \wedge \ldots \wedge \tau_{\lambda_1}(\mathbf{dom}_i)$ où les λ_j sont les variables libres de c. Ainsi peut-on typer chaque n-uple simple $(c_1, \ldots, c_n)$ de $\mathfrak{C}$ en lui rajoutant les informations de types :

$$\tau(c_1, \ldots, c_n) = (c_1, \ldots, c_n) \sqcup \tau_1(c_1) \sqcup \ldots \sqcup \tau_n(c_n)$$

On peut ainsi typer chaque élément de $\mathfrak{C}$ en remplaçant dans chaque formule un n-uple par son homologue typé.

On peut maintenant définir une application de $\mathfrak{C}$ vers $\mathfrak{C}*$ qui à chaque contrainte associe son écriture typée en appliquant les deux règles suivantes :

$$\tau(a \sqcup b) = \tau(a) \sqcup \tau(b)$$

$$\tau(a \sqcap b) = \tau(a) \sqcap \tau(b)$$

Le résultat est la fonction de typage τ :

$$\begin{array}{rccc} \tau : & \mathfrak{C} & \longrightarrow & \mathfrak{C}* \\ & c & \longmapsto & \tau(c) \end{array}$$

Il est facile de voir que la fonction τ envoie sur **faux** toutes les contraintes qui partagent des variables entre des composantes.

Maintenant en se rappelant que $\tau_X(t) \sqcup \tau_X(t') = \tau_X(t \curlywedge t')$ et que $\tau_X(\bot) = \mathbf{faux}$, on peut considérer l'image $\tau(\mathfrak{C})$ de $\mathfrak{C}$ par τ. Celle-ci est isomorphe au sous-ensemble de $\mathfrak{C}$ qui ne partage pas de variables.

On peut remarquer que dans cette construction, le typage rend inconsistant toutes les bases de faits qui partagent des variables entre composantes d'un même vecteur de contraintes. Maintenant, on peut construire un « re-nommage des variables » qui empêche ce cas d'arriver. La construction est la suivante. Chaque ensemble de variables $\mathcal{V}_i$ est coupé en deux parties infinies dénombrables $\mathcal{V}_i^u$ et $\mathcal{V}_i^a$. Maintenant, on sépare $\mathcal{V}$ en n parties infinies dénombrables $\mathcal{V}^1, \ldots, \mathcal{V}^n$. Tous les ensembles utilisés étant de cardinalités égales, on peut choisir sans problème des bijections de l'un vers l'autre. A partir de là, on construit les bijections de re-nommage suivantes comme indiquées dans la figure III-1.

Ainsi, toutes les contraintes de $\mathfrak{C}_i$ utiliseront les variables de $\mathcal{V}_i^u$ (soit $\mathcal{V}^i$) et abandonneront les variables de $\mathcal{V}_i^a$ (soit $\mathcal{V}^j, i \neq j$) pour les autres systèmes. Grâce à cette méthode d'attribution des variables, on n'a jamais de partage de variables entre les systèmes de contraintes.

Pour conclure, on considérera le fragment pleinement typé du produit cartésien saturé comme étant le « bon » produit cartésien de systèmes. Dans ce produit fragment, toutes les

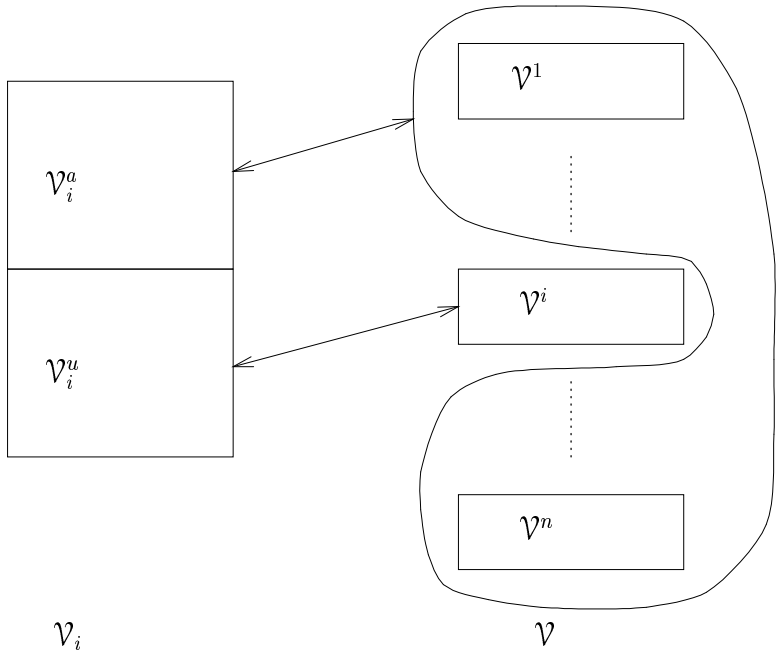


FIG. III-1: Bijection entre $\mathcal{V}_i$ et $\mathcal{V}$

variables des différentes composantes sont disjointes (grâce au système d'allocation de variables précédent). D'autre part, celui-ci n'utilisera jamais des opérateurs diagonaux portant sur deux variables typées différemment. C'est ce fragment-là que l'on nommera le produit cartésien de systèmes de contraintes cylindriques. De surcroît, en utilisant l'attribution des variables préconisées dans la section précédente, il n'y a jamais de conflit entre contraintes autour des variables. On pourra omettre les contraintes de type et ne considérer que $\mathfrak{C}$ au lieu de $\tau(\mathfrak{C})$. Le tout donne le théorème suivant :

THÉORÈME III-4. (PRODUIT CARTÉSIEN DE SYSTÈMES DE CONTRAINTES)
Soit $(\mathfrak{C}_i)_{i\in[1..n]}$, une famille finie de systèmes de contraintes cylindriques, on est capable de lui associer un produit cartésien qui correspond au fragment pleinement typé du produit cartésien saturé typé des systèmes de contraintes de la famille.

1.2.5. RELATION DE CONSÉQUENCE LOGIQUE SUR LE PRODUIT CARTÉSIEN

Il ne reste plus qu'à caractériser la relation de conséquence logique sur le produit cartésien de systèmes de contraintes. Pour la mettre en évidence, il suffit de considérer un produit cartésien de dimension 2. On a donc des paires d'éléments. Maintenant supposons que l'on ait la règle suivante :

$$x, x' \vdash x''$$

Alors

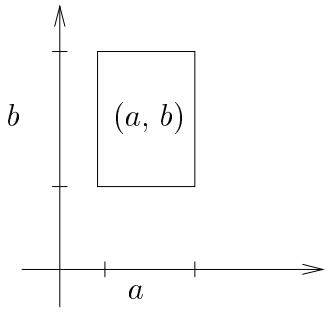
$$\begin{array}{ll} & (x,y) \sqcup (x',y') \\ = & (x,y) \sqcup (x, y \sqcap y) \sqcup (x',y') \sqcup (x', y \sqcap y') \\ = & (x,y) \sqcup (x',y') \sqcup (x \sqcup x', y \sqcap y') \\ \vdash & (x'', y \sqcap y') \end{array}$$

La forme générale devient donc pour un produit cartésien de dimension n :
si $x_i^1, \ldots, x_i^p \vdash y_i$ alors

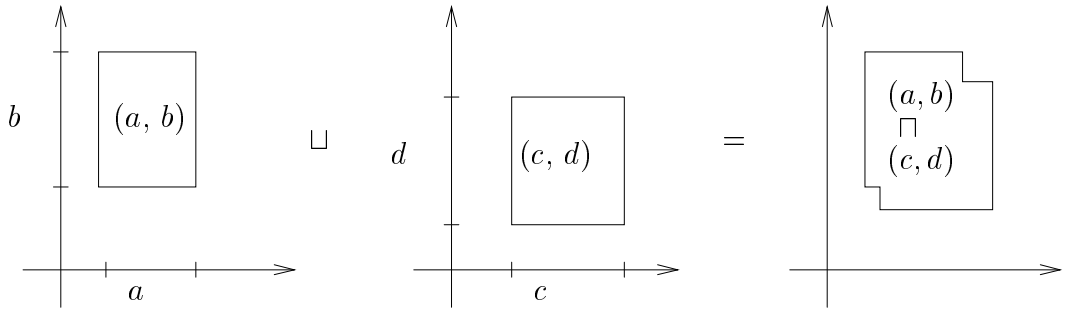
$$(x_1^1, \ldots x_n^1) \sqcup \ldots \sqcup (x_1^p, \ldots, x_n^p) \vdash (x_1^1 \sqcap \ldots \sqcap x_1^p, \ldots, y_i, \ldots, x_n^1 \sqcap \ldots \sqcap x_n^p)$$

1.3. Interprétation graphique

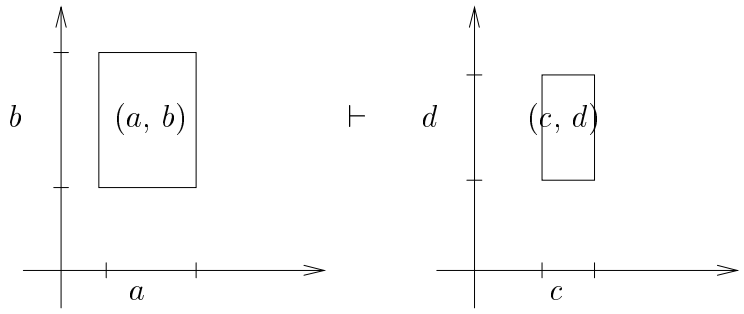
On se propose dans cette section d'illustrer les résultats précédents par quelques graphes explicatifs afin de bien comprendre le produit cartésien de systèmes de contraintes. Pour cela, on va considérer le produit cartésien de deux systèmes de contraintes. Le premier système sera représenté par la droite des abscisses et le second par la droite des ordonnées. De plus, on représentera une contrainte comme un sous-ensemble d'un droite. Dans cet exemple, on considérera des segments. Ainsi une contrainte du produit cartésien sera-t-il une région du plan.



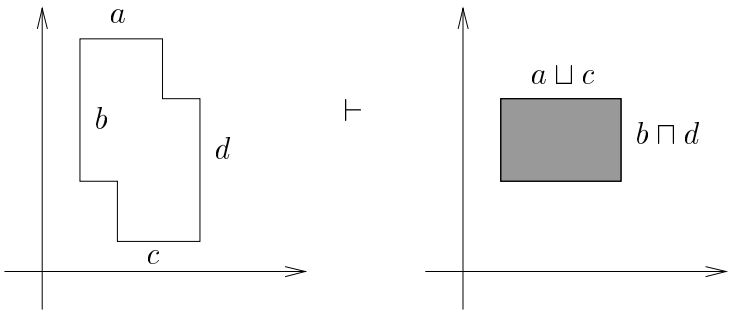
Dans ces conditions, la conjonction de deux couples est l'union des deux régions :

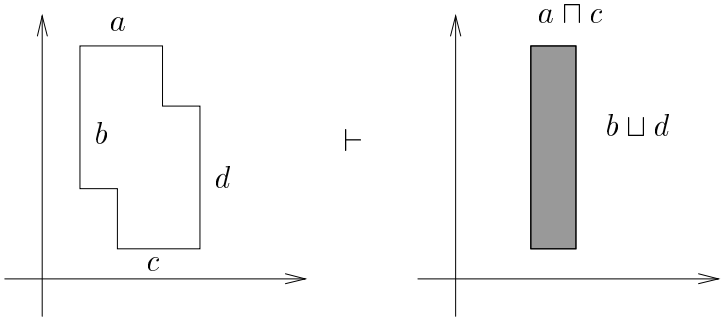


Si l'on s'intéresse maintenant à la relation de conséquence logique, celle-ci se traduit par une inclusion : une contrainte en implique une autre si la représentation de la première contient celle de la seconde :



Dans ces conditions, on peut relier la relation de conséquence logique sur les couples avec les relations de conséquence logique sur les composantes :

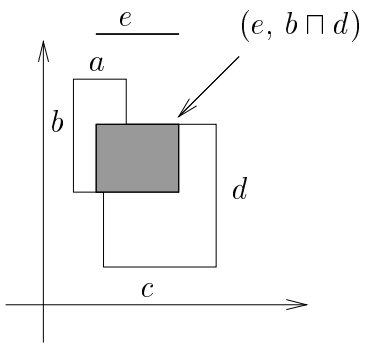




$$(a,\, b) \sqcup (c,\, d)$$

Donc, si $(a \sqcup c \vdash e)$ alors

$$(a, b) \sqcup (c, d) \vdash (a \sqcup c, b \sqcap d) \vdash (e, b \sqcap d)$$



2. La décoration

La décoration est une forme d'étiquette attachée à chaque contrainte. Elle ressemble à une forme générale d'ATMS [43, 34]. De par sa nature, elle doit offrir certaines fonctionalités. La décoration doit pouvoir supporter une forme de comparaison (ordre partiel) et une forme de combinaison (plus petit majorant). La décoration doit donc être au moins un sup-demi treillis. De plus, il doit être possible de vérifier si une décoration est valide, donc ce demi-treillis est muni d'une fonction de validation compatible avec les opérations sur ce demi-treillis.

2.1. Définition et propriétés simples

2.1.1. Définition

Définition III-5. (ordre de décoration)
Un *ordre de décoration* est un sup-demi treillis $\langle \mathcal{O}, \preceq, \curlywedge, \top, \bot \rangle$ muni d'une *fonction de validation* $\zeta : \mathbf{C} \to \{\text{vrai}, \text{faux}\}$ et d'une fonction de composition $\psi : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ qui vérifient les conditions suivantes pour tout $x, y \in \mathbf{C}$:

$$(\zeta(x) = \textbf{faux}) \Rightarrow \zeta(\psi(x, y)) = \textbf{faux}$$

$$(\zeta(x) = \textbf{faux}) \wedge (x \preceq y) \Rightarrow \zeta(y) = \textbf{faux}$$

$$x \leqslant \psi(x, y)$$

$$y \leqslant \psi(x, y)$$

On notera un tel système $\langle \mathcal{O}, \preceq, \curlywedge, \psi, \zeta \rangle$.

Dans cette définition, $\bot$ représente la décoration minimale, $\top$ la décoration incohérente et l'opération $\curlywedge$ représente l'union ou combinaison de deux décorations.

2.1.2. Produit cartésien d'ordre de décorations

La décoration ne devient réellement intéressante que si on peut combiner différents ordres de décoration. C'est le résultat du lemme suivant :

Lemme III-6. (Produit cartésien d'ordres de décoration)
Soit $\langle \mathcal{O}_i, \preceq_i, \curlywedge_i, \psi_i, \zeta_i \rangle_{i \in I}$ une famille finie d'ordres de décoration, on définit le système de décoration produit $\langle \mathcal{O}, \preceq, \curlywedge, \psi, \zeta \rangle$ comme le produit cartésien des demi-treillis (muni des opérateurs produits), muni d'une fonction de validation définie comme la conjonction logique des fonctions de validation sur chacune des composantes.

C'est bien un ordre de décoration.

Avant de passer à la suite, on définit deux familles d'opérations canoniques entre le produit et les composants du produit, à savoir les injections et les projections canoniques.

Définition III-7. (Projections et injections canoniques)
On peut maintenant définir des injections canoniques ξ_i de chaque $\mathfrak{O}_i$ vers $\mathfrak{O}$. Cette injection transforme toute variable d'un système en sa contrepartie typée. Réciproquement, on peut définir des projections canoniques π_i de $\mathfrak{O}$ vers $\mathfrak{O}_i$.

2.2. Exemples simples

2.2.1. Le temps, un ordre total complet

On peut modéliser une information temporelle par la droite des réels positifs complétée par une valeur *fin du temps* ($+\infty$).

On définit l'ordre de décoration $\mathfrak{D}$ suivant :

- les éléments de cet ordre sont les t où $t \in \overline{\mathbf{R}_+}$.
- La relation $\preceq$ est définie par $t \preceq t' \iff (t < t')$
- On définit $t \curlywedge t' = \max(t, t')$.
- $\psi(t, t') = \max(t, t')$
- $\zeta(t) = \textbf{faux} \Leftrightarrow t = +\infty$.

Et la date 0 correspond au début de l'exécution du programme. Elle est aussi le plus petit élément de l'ordre de décoration. La date $+\infty$ est le plus grand élément de cet ordre. C'est la seule décoration invalide. Elle correspond à la fin des temps, soit strictement après toute exécution d'un programme fini.

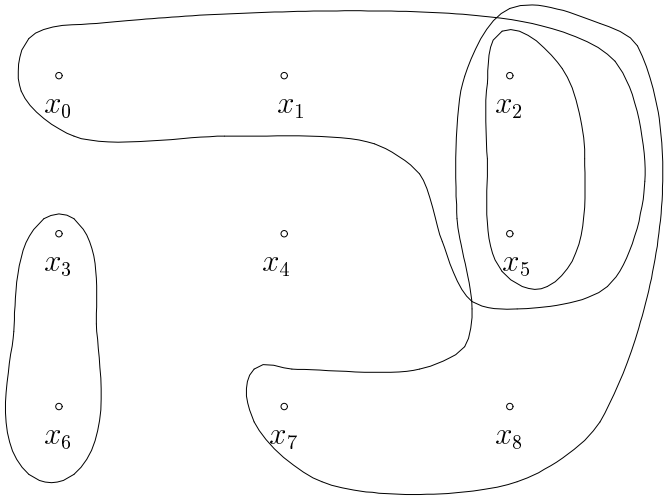
Exemple III-8.: Le temps

2.2.2. Positions, Les parties d'un ensemble

Soit un ensemble dénombrable $\mathbf{S} = \{x_i\}$ de positions, on peut définir l'ordre de décoration $\mathfrak{S}$ suivant :

- Les éléments sont les sous-ensembles de $\mathbf{S}$.
- $\preceq$ est défini par $X \preceq Y \Leftrightarrow X \subset Y$.
- On définit $X \curlywedge Y = X \cap Y$.
- $\psi(X, Y) = X \cap Y$.
- $\zeta(X) = \textbf{faux} \Leftrightarrow X = \emptyset$.

Dans cet exemple, les éléments valides minimaux sont les singletons. $\mathbf{S}$ correspond au plus grand élément, $\emptyset$ au plus petit.



Exemple III-9.: Locations

Cet exemple peut facilement se généraliser à l'ensemble des fermés d'un espace topologique, à un filtre.

2.2.3. Les hypothèses, un ordre partiel quotienté par un ensemble de parties incohérentes

Dans cette exemple, on pousse la comparaison avec les ATMS [43] plus loin. On considère un ordre partiel $(\mathcal{H}, <)$ dénombrable d'hypothèses muni d'un plus grand élément $\top$ et un sous-ensemble `NoGood` de $\mathcal{P}(\mathcal{H})$. On note $2^{\mathcal{H}}$, le treillis des antichaînes de $\mathcal{H}$. Et on définit $\sim$, la plus petite relation d'équivalence qui identifie $\{\top\}$ à toute partie de $2^{\mathcal{H}}$ dominant (dans le sens de l'ordre de domination[2] engendré par $<$) un élément de `NoGood`. L'ensemble $2^{\mathcal{H}}/\sim$ est un ordre décoration.

2.3. Décoration d'un système de contraintes

Il faut maintenant construire un système de contraintes à partir d'un ordre de décoration. A partir de là, décorer un système de contraintes par un système de décoration revient à considérer le produit cartésien des deux systèmes. Il en découle tout naturellement que le produit résultant, muni de la relation de conséquence logique produit est un système de contraintes valide.

2.3.1. Construction du système de décoration

On se donne un ordre de décoration $\langle \mathcal{O}, \preceq, \curlywedge, \zeta \rangle$ muni d'une famille d'opérateurs croissants $(f_i)_{i \in I}$. On se donne aussi un ensemble infini dénombrable de variables $\mathcal{V}$.

Les contraintes du système de décoration sont les contraintes $\diamond(X)$ pour $X \in \mathcal{V}$ et $X = t$ où t est un terme construit avec les f_i, les éléments de $\mathcal{O}$ et les variables de $\mathcal{V}$. On dénote par $\mathfrak{O}$ l'ensemble de ces contraintes clos par quantification existentielle et par conjonction. De plus, on définit sur $\mathfrak{O}$ la relation de conséquence logique $\vdash$ suivante :

$$\diamond(t') \vdash \diamond(t) \Longleftrightarrow t' \preceq t$$

On considère un temps indexé par les réels positifs ou par ∞. On considère aussi les opérateurs croissants $f_i(x) = i + x$ pour $i \geqslant 0$. Dans ce cas-là, les contraintes suivantes sont valides :

- $\diamond(X)$
- $X = f_3(Y) = Y + 3$
- $(\diamond(X) \sqcup X = Y + 2 \sqcup Y = 1) \vdash \diamond(3)$

Exemple III-10.: Système de décoration du temps

---

[2]Une partie H d'un ordre partiel est dominée par une autre H' si et seulement si chaque élement de H possède un majorant dans H'.

2.3.2. Décoration canonique

Un système de contraintes décorées est le produit cartésien d'un système de contraintes par un système de décoration.

Définition III-11. (Décoration canonique)
Soit $\langle \mathcal{C}, \vdash \rangle$ un système de contraintes et $\langle \mathfrak{O}, \sqsubset \rangle$ un système de décoration, on dénote par $\langle \mathcal{C} * \mathfrak{O}$ le produit cartésien du système de contraintes et du système de décoration.

On dénote par Tag, GetConstraint et GetTag les fonctions suivantes :

$$\begin{array}{rccc} \text{Tag}: & \mathcal{C} \times \mathfrak{O} & \longrightarrow & \mathcal{C} * \mathfrak{O} \\ & (c, o) & \longmapsto & \langle c, o \rangle \end{array}$$

$$\begin{array}{rccc} \text{GetConstraint}: & \mathcal{C} * \mathfrak{O} & \longrightarrow & \mathcal{C} \\ & \langle c, o \rangle & \longmapsto & c \end{array}$$

$$\begin{array}{rccc} \text{GetTag}: & \mathcal{C} * \mathfrak{O} & \longrightarrow & \mathfrak{O} \\ & \langle c, o \rangle & \longmapsto & o \end{array}$$

Une contrainte décorée sera dite *valide* si elle est décorée par une décoration valide. Elle sera dite *invalide* dans le cas contraire.

On s'intéresse maintenant à ce que devient la relation de conséquence logique sur le système de contraintes décoré :

Si

$$c_1, \ldots c_n \vdash d$$

alors

$$\langle c_1, o_1 \rangle, \ldots, \langle c_n, o_n \rangle \vdash \langle d, o_1 \sqcap \ldots \sqcap o_n \rangle$$

Donc sous l'hypothèse (qui sera vérifiée par la suite) que chaque o_i est de la forme $\diamond(t_i)$, cette règle devient :

$$\langle c_1, o_1 \rangle, \ldots, \langle c_n, o_n \rangle \vdash \langle d, \diamond(t_1) \sqcap \ldots \sqcap \diamond(t_n) \rangle$$

soit encore

$$\langle c_1, o_1 \rangle, \ldots, \langle c_n, o_n \rangle \vdash \langle d, \diamond(t_1 \curlywedge \ldots \curlywedge t_n) \rangle$$

ce qui est le résultat escompté. Des contraintes décorées interagissent sur la combinaison de leurs décorations.

2.3.3. Propriétés

Le résultat est un système de contraintes valide, c'est ce qu'énonce le théorème suivant.

Théorème III-12. (Système de contraintes décoré)
Soit $(\mathcal{C}, \vdash)$ un système de contraintes et $(\mathfrak{O}, \sqsubseteq)$ un système de décoration.
L'ensemble $(\mathcal{C} * \mathfrak{O}, \vdash')$ est un système de contraintes.

Preuve : C'est une simple conséquence du théorème sur le produit cartésien de systèmes de contraintes. □

REMARQUE III-13. (PRODUIT CARTÉSIEN ET PRODUIT DIRECT)
On pourrait se demander s'il n'aurait pas suffi de considérer le produit direct[3] d'un système de contraintes et d'un système de décoration. La réponse est évidemment non comme le montre l'exemple suivant : Soit $\langle c_1, o_1\rangle$ et $\langle c_2, o_2\rangle$ deux contraintes décorées. La conjonction de ces deux contraintes est bien différente de $\langle c_1 \wedge c_2, o_1 \sqcup o_2\rangle$.

3. Décoration d'un programme CC

Le but de cette section est d'instrumenter un programme CC afin qu'il puisse traiter et manipuler l'information supplémentaire contenue dans la décoration. Cette instrumentation se fait de manière statique, elle rajoute au programme CC des instructions qui vont donc calculer de manière dynamique avec la décoration.

3.1. Décoration compositionelle

Il s'agit donc de transformer un programme CC non décoré en un programme CC décoré. Ce qui est fait de manière compositionnelle avec une décoration courante. De fait, on n'utilisera pas toute la puissance de la décoration et l'on se restreindra à une utilisation purement syntaxique. Dans ce cadre, certains opérateurs seront instrumentés de manière générique, ce sont les opérateurs ***ask***, ***tell***, l'appel de procédure. Sans oublier la décoration initiale qui affecte l'agent initial.

3.1.1. EXEMPLE

On traite ici les opérateurs :***ask***, ***tell***, $\|$, $p(X)$, $p(X) :: A$ et $\exists X$. On traite aussi l'agent initial. Cette instrumentation est en fait une réécriture de la variable courante de décoration qui sera utilisée dans un ***ask*** ou un ***tell***. C'est une simple manipulation syntaxique.

Soit le code suivant : A = $(\textit{tell}(c_1) \parallel p(X))$.

Celui-ci sera instrumenté (avec le langage de décoration $\mathcal{N}$ introduit en fin de chapitre) de la manière suivante :

$$\exists t, (\textit{tell}(\mathsf{tag}(c_1, \mathsf{fork}(t,1))) \parallel p(X, \mathsf{call}(\mathsf{fork}(t,2))))$$

Exemple III-14.: Exemple simple d'instrumentation

3.1.2. NOTATIONS

Pour rendre plus lisible la présentation de l'instrumentation du code, nous utiliserons la syntaxe suivante qui est en fait une règle de réécriture avec contexte :

$$[[(A \| B)]]_t \longrightarrow ([[A]]_{\mathsf{fork}(t,1)} \| [[B]]_{\mathsf{fork}(t,2)})$$

Ce qui signifie que l'instrumentation du code $A\|B$ avec la décoration courante $\diamond(t)$ se réécrit en l'instrumentation récursive des agents A et B avec les décorations courantes qui sont devenues $\diamond(\mathsf{fork}(t,1))$ et $\diamond(\mathsf{fork}(t,2))$.

[3] Dans le produit direct, $(a,b) \times (c,d) = (a \times c, b \times d)$. C'est ce produit la que l'on trouve dans [92].

3.1.3. Agent initial

Un programme CC comporte un agent initial. Celui-ci se retrouve instrumenté avec une décoration initiale t_0 (habituellement $\bot$). Cet agent initial *Init* devient

$$[[Init]]_{t_0}$$

3.1.4. Appel fonctionnel

Pour les fonctions, il faut rajouter l'information de type. Ceci est réalisé grâce à un paramètre en plus.

Si t est la décoration courante, alors on instrumente un appel de fonction de la manière suivante :

$$[[p(X)]]_t \longrightarrow p(X, \mathbf{call}(t))$$

La définition de la fonction est aussi modifiée[4] :

$$[[p(X) :: A]] \longrightarrow p(X, t) :: [[A]]_t$$

3.1.5. Décoration d'un *ask*

Maintenant, il s'agit de traduire le comportement d'un ***ask*** du point de vue de la décoration. En effet, la décoration doit tenir compte de la décoration courante de l'agent qui demande une contrainte, mais aussi de la décoration de la contrainte qui valide l'***ask***. En effet, on veut détecter le cas où le programme continue parce qu'une contrainte invalide a activé un ***ask***. On pourra traiter ce cas par la suite avec la fonction de validité.

Soit le code suivant :

$$[[\mathbf{ask}(c) \rightarrow A]]_{Tag} \longrightarrow \exists_T(\mathbf{ask}(\langle c, \diamond(T)\rangle) \rightarrow [[A]]_{\psi(Tag,T)})$$

Le nouveau code demande une contrainte décorée et recalcule la décoration courante avec l'ancienne décoration courante, la décoration de la contrainte qui active l'***ask*** et à l'aide de la fonction ψ.

Exemple III-15.: Instrumentation d'un ***ask***

Cette dernière expression $\exists_T(\mathbf{ask}(\langle c, \diamond(T)\rangle) \rightarrow [[A]]_{\psi(Tag,T)}$ peut paraître dépasser le cadre de CC standard. En effet, on peut se demander si on n'a pas là affaire à un méta-calcul[5].

Il n'en n'est rien, et la solution se réfère à l'implantation des contraintes indexicales que l'on trouve dans [112]. Comme l'ensemble des décorations est dénombrable (ensemble d'arbres fini avec un nombre fini d'étiquettes), on peut le dénombrer et l'écrire sous la forme

$$\{t_1, \ldots, t_n, \ldots\}$$

[4]Mais il n'y a pas de contexte de décoration dans cette instrumentation puisque la décoration courante est passée en paramètre de l'appel de fonction.

[5]Dans le sens où le calcul ainsi décrit n'est pas exprimable dans le cadre strict des CC et où il nécessite des règles implicites plus puissantes.

Dans ce cas-là, l'expression précédente est une composition parallèle dénombrable d'expressions A_i :

$$A_1 \| \ldots \| A_n \| \ldots$$

où chaque A_i est de la forme

$$\mathbf{ask}(\langle c, \diamond(t_i)\rangle) \rightarrow [[A]]_{\psi(Tag,t_i)}$$

Et cette composition parallèle est bien une expression CC valide.

3.1.6. Décoration d'un *tell*

Un ajout de contrainte utilise finalement la décoration courante :

$$[[\mathbf{tell}(c)]]_t \longrightarrow \mathbf{tell}(\mathrm{tag}(c, \diamond(t)))$$

Cette dernière contrainte utilise donc la décoration courante t, qui est un terme obtenu de manière « syntaxique » depuis la décoration du dernier appel fonctionnel. De plus, cette décoration est obtenue de la même manière depuis l'appel fonctionnel précédent, et ainsi de suite jusqu'à l'agent initial et la décoration initiale. Finalement, la décoration courante est, de manière inductive, toujours équivalente à une décoration close, ce qui est le sujet de la section suivante.

3.2. Décoration fonctionnelle

Si on s'intéresse à la forme des décorations qui apparaissent dans un programme CC décoré, on peut faire la remarque suivante : si la première décoration est un terme clos et si chaque utilisation de la décoration courante n'introduit pas de variable libre supplémentaire, alors celle-ci est toujours équivalente (au sens de la conséquence logique) à un terme clos. La décoration n'est plus relationnelle (les contraintes de décorations sont des relations entre les variables de décoration) mais fonctionnelle (chaque contrainte de décoration est une fonction qui calcule une valeur unique pour chaque variable de décoration).

Plus précisément, on peut donner la définition suivante :

Définition III-16. (Programme décoré clos)
Soit P un programme CC décoré, on dira que P est clos du point de vue de la décoration si les conditions suivantes sont vérifiées :

(i) La décoration initiale est égale à un terme sans variable.

(ii) Chaque contrainte est posée avec une décoration de la forme $\diamond(T)$ où T est une formule ne contenant pour seule variable que la décoration courante.

(iii) Chaque appel de procédure transmet une décoration courante sous la forme

$$NewTag = F(CurrentTag)$$

où F est une formule ne contenant que la seule variable $CurrentTag$ qui est la décoration courante.

Suite à cette définition, prouver qu'un programme est clos du point de vue de la décoration se fait de manière purement syntaxique.

Dans ces conditions, il est facile de prouver par induction sur le programme le lemme suivant :

LEMME III-17. (DÉCORATION FONCTIONNELLE)
Si P est un programme CC décoré qui vérifie la propriété de clôture du point de vue de la décoration, alors chaque décoration qui apparaît dans la base de faits est équivalente à une décoration close.

En pratique, on n'observera que les décorations closes dans les programmes. Dans ce cas, celles-ci sont toujours de la forme $\diamond(t)$ où t est un élément de l'ordre de décoration.

3.3. Décoration atomique

3.3.1. ATOMICITÉ DE LA DÉCORATION COURANTE

On suppose que dès qu'un programme pose une contrainte c (*tell* c) alors il pose $c\downarrow$, c'est-à-dire toutes les contraintes directement impliquées par c[6]. Cette propriété de saturation nous permettra de rendre bijective la relation entre les contraintes posées par le programme et les décorations simples (voir après).

A partir de là, soit la définition inductive suivante :

DÉFINITION III-18. (DÉCORATION ATOMIQUE)
Une décoration sera dite atomique si elle n'utilise pas dans son écriture le symbole $\curlywedge$ sauf dans le deuxième membre d'un lexème $\psi(x, y)$.

On obtient la propriété fondamentale suivante :

THÉORÈME III-19. (LOCALISATION DES DÉCORATIONS ATOMIQUE)
Soit P un programme CCétendu implanté au-dessus de CC avec la méthode précédente et le système de décoration $\mathcal{D}$. Si la décoration initiale est atomique, alors

Prop. 1 : Toute contrainte décorée posée par un programme (*tell* c) l'est avec une décoration atomique.

Prop. 2 : Réciproquement, toute contrainte décorée avec une décoration atomique a été posée par un *tell* du programme.

Preuve : La preuve de la première affirmation est une simple vérification du fait que l'instrumentation d'un programme CC conserve le caractère atomique de la décoration. Donc par simple induction, il est facile de se convaincre que si la décoration initiale est atomique, alors toutes les décorations utilisées par le programme lors de son exécution seront atomiques.

Réciproquement, toute contrainte impliquée par l'ajout d'une contrainte c à la base de faits est conséquence d'au moins deux contraintes (par saturation du *tell* c). D'autre part, toute décoration associée à une opération d'ajout (*tell*) est une feuille de l'arbre de décoration (par terminalité du *ask*) donc elle est incomparable avec toute autre décoration associée à

[6]Les contraintes directement impliquées par c sont celles impliquées dans toute base de faits compatible avec c.

un ajout de contrainte. Ce qui permet d'affirmer que la décoration associée à une contrainte impliquée par la base de faits est toujours une combinaison de décorations incomparables. Donc la décoration de cette contrainte impliquée utilise le symbole $\curlywedge$. □

De plus, comme l'opération $\sqcap$ est associative et commutative, toute décoration de la base de faits s'écrit comme une disjonction finie de décorations atomiques :

$$d_1 \sqcap \ldots \sqcap d_n$$

En outre, comme l'implantation décorée d'un programme CCétendu vérifie la propriété de clôture du point de vue de la décoration (définition III-16.), alors chaque décoration de la base de faits est équivalente à une décoration close (lemme III-17.). Soit de la forme :

$$\diamond(t_1) \sqcap \ldots \sqcap \diamond(t_n)$$

ce qui est équivalent à

$$\diamond(t_1 \curlywedge \ldots \curlywedge t_n)$$

En pratique, on n'observera que les contraintes décorées par une décoration de cette forme. Ce qui ne limite en rien l'étude, toujours d'après le lemme III-17..

3.3.2. Lecture d'une base de faits saturée

Donc, maintenant qu'il y a cette propriété d'unicité, on peut interroger la base de faits d'après les décorations atomiques ou d'après les contraintes posées par le programme. Cela nous servira de la manière suivante : on se demandera si une contrainte e (avec le programme qui a posé ***tell*** d et $c, d \vdash e$ est licite, pour cela, il convient de se demander si la contrainte d a réagi avec la bonne instance de c, on interrogera alors la base pour connaître toutes les contraintes de la classe de c, et pourra choisir celle qui est sélectionnée pour réagir avec d, ainsi les contraintes impliquées par cette bonne instance de la classe de c seront licites et toutes les autres seront illicites.

Il reste à savoir si un CC ainsi décoré calcule au moins autant qu'un CC non décoré, C'est une propriété de saturation de la base de fait. Nous utilisons un analogue de la propriété de redémarrage des agents, qui dit que dans un CC déterministe, on peut relancer un agent autant de fois qu'on le désire sans changer le résultat. Nous supposerons donc que le programme explore dans ses calculs toutes les combinaisons possibles de réveil d'un ***ask*** par une contrainte afin d'être sûr que le programme exécute un réveil valide d'un ***ask*** s'il en existe un.

3.4. Contrôle de l'exécution par la décoration

On s'intéresse maintenant à la sélection des contraintes valides dans une base de faits décorée. On rappelle qu'une contrainte décorée est valide si la fonction de validation ζ appliquée à sa décoration est égale à **vrai**.

Cette opération de sélection est en fait une opération d'effacement. Il faut vérifier que le résultat est bien une base de faits cohérente. Pour cela, regardons les conséquences qu'a pu avoir une contrainte invalide sur la base de faits. Celles-ci sont de deux natures, soit elle a réagi avec des contraintes pour en générer d'autres, soit elle a autorisé un ***ask*** à se réduire.

3.4.1. Implications d'une contrainte invalide

Le traitement de ce cas est simple. Toute conséquence d'une contrainte invalide est une contrainte invalide comme l'énonce le lemme suivant :

Lemme III-20.
Si $\langle c_1, o_1\rangle, \ldots, \langle c_n, o_n\rangle \vdash \langle c, o_1 \sqcap \ldots \sqcap o_n = o\rangle$ **et si** $(\exists i \mid \zeta(o_i) = \textbf{faux})$ **alors** $\langle c, o\rangle$ **est invalide.**

Preuve : Tout simplement parce que $o_i \vdash o_1 \sqcap \ldots \sqcap o_n$ et donc par définition, $\zeta(o) = \textbf{faux}$. □

3.4.2. Validité et opérateur *ask*

Là encore le résultat est une conséquence de la définition de ζ. Comme ζ est compatible avec la fonction ψ, si une contrainte invalide active un ***ask***, toute contrainte posée par l'agent qui suit l'***ask*** est invalide.

De plus, quand ψ est défini a priori, on peut aussi introduire un test à l'exécution de l'***ask***. Le code

$$\textit{ask}(c) \rightarrow A$$

devient donc (toujours avec Tag étant la décoration courante),

$$\exists T(\textit{ask}(\langle c, \diamond(T)\rangle \sqcup \zeta(\psi(Tag, T)) = \textbf{vrai}) \rightarrow (NewTag = \psi(Tag, T) \| A))$$

3.4.3. Nature de la sélection

Si l'on s'intéresse maintenant à la forme de la base de faits finale B , celle-ci se présente comme une conjonction de contraintes décorées. Cette conjonction peut se réécrire sous la forme $V \sqcup N$, où chaque contrainte de V est valide, et où chaque contrainte de N est invalide. On notera

$$\zeta(B) = V$$

pour désigner la sélection de la base de faits finale par la fonction de validité ζ associée au système de décoration utilisé.

4. Application à la sémantique du non-déterminisme

Nous étudions dans cette partie la manière dont la décoration peut être utilisée pour plonger un calcul CC non-déterministe angélique dans un calcul CC déterministe décoré.

4.1. Un opérateur $\oplus$ pour le non-déterminisme

Nous introduisons un nouvel opérateur $\oplus$ différent du non-déterminisme usuel car dans le contexte des CC, ce dernier souffre de problèmes qui le rendent peu adapté au langage $\texttt{CC}(\mathcal{M})$.

4.1.1. Motivations

Si la définition usuelle du non-déterminisme angélique[63] (en anglais, « don't know non-determinism ») est adaptée aux langages logiques en général, elle souffre de problèmes sérieux dans le cadre des CC. En effet, on peut distinguer deux lacunes sérieuses dans sa définition.

La première est mise en relief dans [72], elle montre que la combinaison du non-déterminisme (angélique), de l'indéterminisme et de la composition parallèle est difficile à caractériser de manière claire. En effet, on peut obtenir avec un programme très simple des comportements peu intuitifs[7]. Or, comme on le verra dans sa définition, le langage CC($\mathcal{M}$) utilise le non-déterminisme et la composition parallèle dans sa définition. De plus, il est naturellement indéterministe. Par conséquent, il est vital de lui donner une sémantique plus simple et plus claire.

Le deuxième problème est lié à ce que l'on appelle le choix aveugle et le choix gardé. Le non-déterminisme usuel dans le cadre des CC fait appel au choix gardé. C'est-à-dire que toutes les branches d'un point de choix sont gardées par des clauses de test. Or ce choix gardé est mal adapté à au langage CC($\mathcal{M}$) qui, de par la nature même des problèmes d'optimisation combinatoire auquel il est destiné, a besoin d'un choix aveugle (sans garde). Grosso modo, les algorithmes de résolution de problèmes de contraintes ont besoin d'heuristiques très complexes de choix de la branche à explorer. Ces algorithmes sont si complexes qu'il deviennent difficiles à implanter dans un langage avec gardes. On a besoin d'un système non-déterministe et non d'un langage gardé [8] [25, 26, 24].

Enfin, on peut aussi s'attacher à donner une sémantique du non-déterminisme plus proche de son implantation dans un langage concurrent équitable. On mettra l'accent tout particulièrement sur son modèle d'exécution qui devra refléter la concurrence vraie qui est l'une des idées fondatrices du projet CC($\mathcal{M}$).

4.1.2. Analyse de l'opérateur $\oplus$

Analyse

L'idée première derrière l'opérateur $\oplus$ est la suivante. Un calcul non-déterministe est un calcul concurrent (ou effectué par une co-routine) dont les résultats sont masqués du reste du calcul. En d'autre termes, si, grâce à la décoration, on est capable d'isoler entre eux les deux branches d'une composition parallèle de deux agents, alors il suffit de collecter les résultats de ces deux calculs pour obtenir le résultat non-déterministe de ces deux agents. C'est ce que l'on appelle la recherche encapsulée[95, 96]. On se propose de décrire ce mécanisme au sein même des CC avec la décoration.

L'idée principale vient du parallélisme de données « ou » [101]. Il reste donc à isoler deux branches (ou plus) d'un même opérateur de composition parallèle l'une de l'autre et d'interdire les interactions entre des branches traduisant des hypothèses distinctes à l'aide de la fonction de validation ζ associée à la décoration. Il reste une question : comment propager

[7] Dans l'article, on peut voir comment l'ordre de réduction des opérateurs influe sur le résultat des calculs, jusqu'à obtenir dans un même calcul les deux branches distinctes d'un point de choix indéterministe.

[8] Comme on peut le voir dans [102], les stratégies de recherche de points de choix n'utilisent pas les gardes mais un processus de recherche et d'instantiation des points de choix.

une information qui permette d'identifier sans ambiguïté à quelle place de l'arbre de recherche se trouve chaque agent lorsqu'il interagit avec la base de faits (par l'ajout ou la demande d'une contrainte) ?

On peut fournir une réponse simple : si on trace l'arbre d'exécution des agents construit avec les constructeurs des `CC`, alors on peut identifier précisément le chemin (la trace) qu'a suivi un agent avant d'interagir sur la base de faits. En outre, il est clair que l'appel de procédure et la quantification existentielle, qui sont des constructeurs unaires au niveau des agents, n'introduisent pas d'ambiguïté quand à la place des agents dans l'arbre de choix. On peut donc les oublier dans cette même trace. Finalement, pour situer les agents, on a besoin de deux constructeurs, un pour la composition parallèle, un pour le non-déterminisme. A ceci, on rajoute un constructeur pour l'*ask*.

Syntaxe

On se propose donc d'introduire l'opérateur `CC` suivant :

Agent	: :=	...	
	|	$\oplus_i$ Agent$_i$	*Non-déterminisme aveugle*

4.2. Implantation de l'opérateur $\oplus$

L'opérateur $\oplus$ est implanté grâce à la décoration des `CC` avec un ordre de décoration idoine.

4.2.1. Un langage de décoration pour le non-déterminisme

Les termes du langage

On définit le langage de décoration $\mathcal{N}$ suivant comme une algèbre de termes avec variables avec les constructeurs suivants :

- **unify** : $\mathcal{N} \times \mathcal{N}$ pour la fonction ψ.
- **fork** : $\mathcal{N} \times \mathbf{N}$ qui servira à décorer la composition parallèle.
- **branch** : $\mathcal{N} \times \mathbf{N}$ qui servira à décorer l'opérateur $\oplus$.
- $\top$ pour la décoration minimale.
- $\bot$ pour la décoration incohérente.

L'ensemble de termes considéré est la clôture de ces termes par l'opération associative commutative $\curlywedge$.

Constructeurs du langage

De plus on associe à chaque constructeur du langage de $\mathcal{N}$ les opérations croissantes de même nom qui permettent de construire les termes :

On définit :

$$\begin{array}{rccc} \textbf{branch}: & \mathcal{N} \times \mathbf{N} & \longrightarrow & \mathcal{N} \\ & (t, n) & \longmapsto & \textbf{branch}(t, n) \end{array}$$

Exemple III-21.: Opérateur **branch**

Ordonnancement des décorations

On définit l'ordre sur les décorations $\sqsubseteq$ comme la plus petite relation d'ordre qui vérifie les règles suivantes :

Règle 1 : $\forall n > 0, \forall t_i \in \mathfrak{D}, t_j \preceq \curlywedge_{i<n}(t_i)$.

Règle 2 : $\forall t_1, t_2 \in \mathfrak{D}, (t_1 \preceq \mathbf{unify}(t_1, t_2)) \wedge (t_2 \preceq \mathbf{unify}(t_1, t_2))$.

Règle 3 : $\forall t \in \mathfrak{D}, \forall i \in \mathbf{N}, t \preceq \mathbf{branch}(t, i)$.

Règle 4 : $\forall t \in \mathfrak{D}, \forall i \in \mathbf{N}, t \preceq \mathbf{fork}(t, i)$.

On aurait pu réécrire les quatre dernières règles par la règle suivante : Si t_1 est un sous-terme de t_2 alors $t_2 \sqsubseteq t_1$.

Fonction de validation ζ

Enfin, on définit la fonction de validation ζ en posant $\zeta(t) = \mathbf{faux}$ si et seulement si il existe t' dans $\mathcal{N}$ et i, j deux entiers distincts tels que $\mathbf{branch}(t', i)$ et $\mathbf{branch}(t', j)$ soient tous les deux des sous-termes de t . Dans l'alternative, $\zeta(t) = \mathbf{vrai}$.

Le cas où la fonction de validation ζ devient égale à **faux** correspond à l'existence d'un point de choix (symbolisé par t') présent deux fois dans un même calcul (symbolisé par t) avec des hypothèses distinctes (symbolisées par i et j).

4.2.2. Instrumentation d'un programme avec $\mathcal{N}$

L'opérateur $\exists_X$

Cet opérateur `CC` n'a pas d'influence sur la décoration courante, donc :

$$[[\exists_X A]]_t \longrightarrow \exists_X [[A]]_t$$

Composition parallèle

Si A_i est la i-ème branche d'un opérateur de composition parallèle et si t est la décoration courante avant le $\|$ alors la décoration courante devient $\mathbf{fork}(t,\ i)$.

$$[[A_1 \| \ldots \| A_n]]_t \longrightarrow [[A_1]]_{\mathbf{fork}(t,1)} \| \ldots \| [[A_n]]_{\mathbf{fork}(t,n)}$$

*Opérateur **ask***

Pour instrumenter un ***ask***, on utilise le constructeur **unify** qui correspond à la fonction ψ de combinaison de décoration :

$$[[\textbf{\textit{ask}}(c) \to A]]_t \longrightarrow \exists_{t'} (\textbf{\textit{ask}}(\langle c, t' \rangle) \to [[A]]_{\mathbf{unify}(t,t')})$$

Implantation de l'opérateur $\oplus$

On implante en deux temps. Dans un premier temps, on implante syntaxiquement l'opérateur $\oplus$ en parallélisant des agents avec des termes représentant des hypothèses disjointes :

$$[[A_1 \oplus \ldots \oplus A_n]]_t \longrightarrow [[A_1]]_{\mathbf{branch}(t,1)} \| \ldots \| [[A_n]]_{\mathbf{branch}(t,n)}$$

Le deuxième temps est un temps de sélection dans une base de faits finale. Ce deuxième temps est traité dans la section suivante.

4.2.3. Fonction de validation associée à l'opérateur $\oplus$

Une fois que les calculs sont effectués et que la base de faits est arrivée dans un état final, on peut l'observer et la trier afin d'en éliminer les contraintes qui sont associées à des décorations invalides. Cette sélection entraîne plusieurs remarques.

La première est que toute combinaison (selon la règle de conséquence logique) de contraintes issues d'hypothèses différentes est obligatoirement illicite, de par la définition même de ζ. On a donc réussi à isoler des branches de calcul disjointes en rendant invisibles leurs interactions.

La deuxième remarque porte sur les agents en attente de contraintes (***ask***). Ceux-ci peuvent être réveillés par une contrainte illicite. Leurs actions sont par la suite effacées de la base de faits finale par ζ. Mais ce critère peut être localisé au niveau même du code instrumenté. En effet, si on remplace l'instrumentation d'un ***ask*** par :

$$[[\textbf{\textit{ask}}(c) \to A]]_t \longrightarrow \exists_{t'}(\textbf{\textit{ask}}(\langle c, t'\rangle \wedge (\zeta(\textbf{unify}(t, t')))) \to [[A]]_{\textbf{unify}_{(t,t')}})$$

Alors les agents en suspends ne peuvent être activés que par des contraintes licites. Cette application illustre bien le pouvoir d'expression de la décoration qui permet d'exprimer de manière globale des algorithmes de contrôle local.

4.3. Résultats

Dans cette section, nous vérifions si l'opérateur $\oplus$ répond aux attentes formulées dans la section 4.1.1. et nous le comparons au non-déterminisme usuel.

4.3.1. Propriétés de l'opérateur $\oplus$

Combinaison non-déterminisme, indéterminisme et composition parallèle

Si l'on reprend l'exemple de l'article[72] :

$$(c_1 + c_2) \| (c_3 \vee c_4)$$

où $+$ représente la composition indéterministe et $\vee$ la composition non-déterministe. Avec un opérateur usuel, on peut obtenir dans un même calcul $\langle \{c_1, c_3\}, \{c_2, c_4\}\rangle$, soit c_1 et c_2 dans un même résultat, ce qui semble absurde. Tandis qu'avec l'opérateur $\oplus$ à la place de $\vee$, comme celui-ci ne reconstruit pas une base de faits finale commune, on obtient une base de faits comportant c_1 ou c_2 ainsi que deux bases de faits locales contenant c_3 et c_4. Ce qui est plus intuitif. On a donc réussi à définir l'opérateur non-déterministe maximal qui garde un comportement simple en présence de l'indéterminisme et de la composition parallèle.

Choix aveugle

L'opérateur $\oplus$ implante visiblement un choix aveugle. Ce qui permet à l'agent **stop** d'être élément neutre pour cet opérateur. De plus la composition parallèle et l'opérateur $\oplus$ commutent entre eux (ce qui se traduit par un re-numérotage des arguments numériques des constructeurs **fork** et **branch**).

Modèle d'exécution

Finalement, l'opérateur $\oplus$ nous indique la voie pour une intégration plus souple du non-déterminisme dans les langages concurrents par contraintes. En effet, le non-déterminisme est vu non plus comme un retour en arrière mais comme un calcul en avant. Il est présenté ici de manière formelle comme une extension conservative des `CC` déterministes grâce à la décoration. Mais cette approche a l'avantage de ne pas différencier la composition parallèle et la disjonction pour le gestionnaire de tâches (« scheduler »). Ce qui simplifie le modèle d'exécution.

4.3.2. Comparaison avec le non-déterminisme usuel

Comme pressenti dans la section précédente, l'opérateur $\oplus$ n'explore pas, comme en Prolog, un arbre de choix de manière séquentielle en collectant les résultats au fur et à mesure. Au contraire, il explore cet arbre en parallèle et stockant les résultats sur place dans des bases de faits locales. De là vient toute la différence entre l'opérateur $\oplus$ et le non-déterminisme classique puisque ce premier est une parallélisation de ce dernier. Par contre, on ne s'intéresse pas du tout à des évolutions du schéma du non-déterminisme telles que le partage de structures, la tabulation [30].

Ce modèle sera étendu par la suite en introduisant des variables communes (qui ne seront pas décorées par la position de l'agent dans l'arbre de recherche). Grâce à ces variables globales (variables décorées par la décoration minimale), il sera facile d'implanter un collecteur de résultats à la manière d'un opérateur **bagof** comme on peut le trouver dans le langage AKL[64].

Chapitre IV

Les variables impératives en CC

Comme nous avons pu le voir dans l'introduction, l'utilisation du langage CC($\mathcal{M}$) pour implanter des domaines de contraintes requiert une certaine forme de non-monotonie dans sa spécification. Celle-ci prend la forme de variables impératives sur lesquelles on peut écrire de manière destructive.

Pour décrire cette extension des CC, il est suffisant de raisonner sur la trace d'exécution d'un programme CC. En effet, à l'aide d'un algorithme de sélection approprié, il est possible de choisir dans une base de faits finale décorée par des informations de trace, les exécutions qui n'ont tenu compte que des valeurs réelles des variables au moment de leur lecture.

Les variables impératives se réduisent donc à une sélection sur une base de faits décorée par une information suffisante pour tracer dans le temps l'exécution d'un programme : ajout d'une contrainte, demande d'une contrainte.

1. Présentation d'une extension impérative des CC

1.1. Deux approches pour les variables impératives

1.1.1. Les variables flots

On peut trouver une première technique d'implantation des variables impératives avec les variables flots (en anglais *stream*). On peut trouver une trace de cette idée dans [67]. Une autre implantation est celle des ports dans AKL [64].

Cette méthode est basée sur des listes terminées par une variable non instanciée :

$$X = V_1 :: \ldots :: V_n :: Y$$

où les V_i sont les valeurs successives de la variable X. L'écriture et la lecture sont basées toutes les deux sur un mécanisme indéterministe de descente récursive le long de la liste des valeurs de X.

$$\mathbf{write}(X, v) = (\exists Z, X = v :: Z) \mid (\mathbf{write}(tail(X), v);$$

$$\mathbf{read}(X, Y) : -(Y = head(X)) \mid (\mathbf{read}(tail(X), Y);$$

Dans les deux cas, la bonne exécution est celle qui descend récursivement la liste des valeurs de X jusqu'à la dernière cellule instanciée et qui soit la lit, soit écrit à la suite. Mais ce test de fin de liste est équivalent à un test $var(X)$, qui répond si X est instancié ou non. Ce test est clairement non-monotone.

1.1.2. Les cellules

On peut trouver une autre technique d'implantation des variables impératives avec les cellules [105, 73, 103]. Cette méthode propose deux constructeurs `NewCell` et `Exchange`.

Les cellules sont créées par la commande

$$\mathsf{NewCell}(X, Y)$$

Cette commande choisit un nouveau nom ξ, lie X à ξ et stocke la cellule $\xi : Y$ en mémoire. Puis, la lecture de cette cellule se fait par la commande

$$\mathsf{Exchange}(X, U, Z)$$

Cette commande attend que la variable X soit liée à un nom ξ et que l'on ait la cellule $\xi : Y$. Dans ce cas, la cellule est changée en $\xi : U$ et la contrainte $Z = Y$ est ajoutée à la base de faits.

Le principal intérêt des cellules est que la lecture et l'écriture sont synchronisées et atomiques. Il n'y a pas de problèmes de sections critiques dans un environnement concurrent ou parallèle avec les cellules.

Par rapport à la méthode des variables flots, les cellules maintiennent une référence non monotone à la queue de la liste des valeurs. Ainsi il n'y a plus besoin d'un contrôle local sur la lecture et l'écriture. Mais en revanche, les cellules utilisent des affectations destructives sur les valeurs des cellules.

1.2. Les variables impératives de CC($\mathcal{M}$)

1.2.1. Un contrôle basé sur la décoration

Les variables impératives se présentent dans le langage CC($\mathcal{M}$) comme une reformulation du principe des variables flots à l'aide de la décoration. Dans cette reformulation, pour affecter une valeur à une variable, au lieu d'instancier la fin de la variable flot, on rajoute une nouvelle contrainte d'écriture de la valeur sur la variable et, lors d'une lecture, on choisit la *bonne* valeur de manière globale avec la décoration.

Pour cela, si chaque contrainte de lecture et d'écriture est décorée avec une information de date, privilégier la bonne interaction entre des contraintes de lecture et d'écriture revient à choisir entre toutes les interactions possibles celles qui sont optimales du point de vue de la date, c'est-à-dire que la lecture retourne la dernière valeur écrite.

Il reste à spécifier comment dater chaque action du programme. En particulier, il faut proposer un système qui tienne compte de l'ordre d'exécution des agents entre eux. Un tel système ne peut se satisfaire d'une information de type horaire puisque celle-ci est fonction de l'exécution du programme. On décide donc de dater le programme par une information de position dans l'arbre syntaxique d'exécution du programme. Cet arbre sera ensuite plongé dans un ordre total possible qui représentera une ordre d'exécution possible.

L'intérêt de cette méthode est double :

Tout d'abord, la technique employée est compatible avec l'implantation du non-déterminisme avec l'opérateur $\oplus$. Ce qui permet de décrire simplement l'interaction entre des variables impératives et des systèmes non-déterministes.

Enfin, étant donné que la datation des opérations de lectures et d'écritures est basée sur l'entrelacement des agents, on peut établir précisément la correspondance entre le résultat calculé et le choix de l'ordre d'exécution des agents.

1.2.2. Un domaine simple de lecture-écriture sur des variables

On considère un domaine de contraintes très simple qui possède deux atomes. Le premier **write** prend deux arguments, une variable et un entier et correspond à l'écriture sur cette variable de la valeur entière. Le second **read** prend deux variables comme argument et correspond à l'écriture sur une variable de la valeur lue sur l'autre. Ce domaine possède une seule règle d'inférence :

$$\mathbf{write}(X, a),\ \mathbf{read}(Y, X) \vdash \mathbf{write}(Y, a)$$

Si on analyse bien les besoins du langage $\mathtt{CC}(\mathcal{M})$, on a besoin de pouvoir spécifier $\mathbf{write}(X, 1)$, puis à partir d'un certain moment $\mathbf{write}(X, 2)$, puis $\mathbf{write}(X, 3)$ et ainsi de suite. Une manière de voir cette propriété est de considérer qu'il y a une certaine classe d'équivalence $\backsim$ sur les contraintes (dans ce cas particulier $\forall a, b, (\mathbf{write}(X, a)) \backsim (\mathbf{write}(X, b))$) et qu'il ne peut y avoir deux instances valides en même temps d'une même classe d'équivalence. Donc poser $\mathbf{write}(X, a)$ *écrase* la précédente contrainte attachée à X.

Cette opération est un exemple très simple de non-monotonie. Elle n'implique pas le retrait d'agent ou autre opération sur la base de faits comme on le trouve dans la littérature [41, 9, 89].

1.2.3. Un opérateur de séquentialité

L'écrasement de contraintes ne devient utilisable que si l'on est capable de préciser l'ordre dans lequel les contraintes s'écrasent les unes les autres. L'ordre induit par les ***ask*** se révèle vite très insuffisant[1]. Pour remédier à ce manque, on introduit un opérateur de séquentialisation ' ;'.

La syntaxe est la suivante :

Agent	: :=	...	*Définitions usuelles*
	|	Agent ; Agent	*Séquence*

[1] Pour se convaincre de cette affirmation, imaginons un programme Prolog vu comme un programme CC sur un système de contraintes basé sur les termes de Herbrand. Un tel programme n'utilise pas l'opérateur ***ask***. Dans un tel langage, les variables impératives sont introduites par l'opérateur **is**. Or, d'après l'utilisation courante de ce langage, il est clair que les programmes utilisent la séquentialisation implicite des buts à résoudre (résolution de gauche à droite). Une forme de séquentialisation est nécessaire.

1.2.4. Deux exemples de programmes parallèles et séquentiels

Programme séquentiel

On reprend le domaine de contraintes précédent et on définit la relation d'équivalence comme la plus petite relation d'équivalence qui vérifie :

$$\forall a, b, (\mathbf{write}(X, a)) \backsim (\mathbf{write}(X, b))$$

Ceci étant fait, on peut donner un premier exemple de programme séquentiel :

Soit le programme suivant,

$$\mathbf{write}(X, 0); \mathbf{write}(X, 1); \mathbf{read}(Y, X); \mathbf{write}(X, 2)$$

On voudrait qu'il ne calcule qu'une seule contrainte : $\mathbf{write}(Y,\ 1)$.

Exemple IV-1.: Programme purement séquentiel

Au moment où la contrainte $\mathbf{read}(Y, X)$ est posée, la seule instance valide de la classe des contraintes $\mathbf{write}(X, v)$, où v est un entier, est $\mathbf{write}(X, 1)$. Cet exemple illustre bien deux notions : la notion de séquence avec écrasement de contraintes (car les écritures successives sur X écrasent les précédentes) et deuxièmement la notion de contraintes prioritaires ou réactives, car la valeur attendue de Y est celle de X juste au moment de l'écriture de l'atome $\mathbf{read}(Y,\ X)$.

Exemple séquentiel et parallèle

Que se passe-t-il quand on mélange séquence et parallélisme. Soit le deuxième exemple suivant :

Soit le programme :

$$(\mathbf{write}(X, 1) \| \mathbf{write}(X, 2)); \mathbf{read}(X, Y)$$

Exemple IV-2.: Programme séquentiel-parallèle

Quelle est l'instance de $\mathbf{write}(X,\ \_)$ qui est valide au moment du **read** ? Quelle est la contrainte générée ?

1.2.5. Lien entre les variables impératives et l'entrelacement des agents

Étudions ce deuxième exemple. On se base d'abord sur un modèle standard de la concurrence [23, 83] qui modélise l'exécution d'un programme par un entrelacement d'actions atomiques. Ici, les actions atomiques sont l'ajout d'une contrainte, le réveil d'un ***ask***, la composition parallèle, l'appel de procédure et la quantification existentielle.

Si on décrit l'arbre des processus dans la figure IV-1

Il y a autant d'exécutions possibles que de plongement de l'arbre dans un ordre total de manière compatible avec la sémantique du calcul de processus. Formellement si l'arbre T a n sommets, on cherche une fonction ρ bijective de T dans [1 .. n] qui vérifie :

Cond 1 : Si le sommet s_1 est un fils de s_2 alors $\rho(s_1) > \rho(s_2)$.

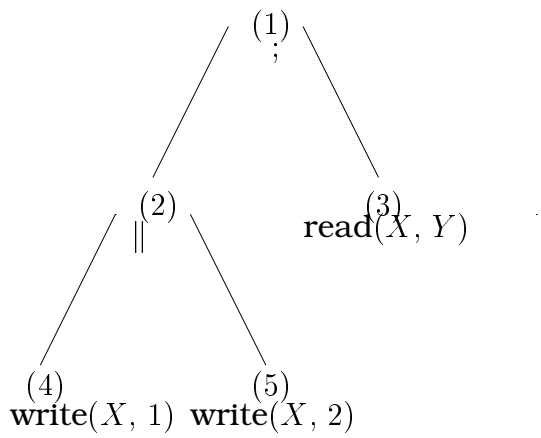


FIG. IV-1: Arbre de processus de l'exemple IV-2.

Cond 2 : Si le sommet s_0 est étiqueté par ';' alors si s_1 et s_2 sont des fils de s_0 et si s_1 est plus à gauche que s_2 alors $\rho(s_2) > \rho(s_1)$. De plus, si s_3 est un descendant de s_1 et s_4 est un descendant de s_2 alors $\rho(s_4) > \rho(s_2)$.

On a les deux ordres possibles suivants :

$$\rho([1,2,3,4,5]) = [1,2,4,5,3]$$

ou

$$\rho[1,2,3,4,5]) = [1,2,5,4,3]$$

Qui correspondent aux deux entrelacements possibles des fils du sommet (2). Dans le premier cas, la contrainte attendue est $Y = 1$ et dans le deuxième, $Y = 2$.

Ce qui se traduit pour la séquence par la règle opérationnelle suivante :

$$\frac{\text{A} \longrightarrow \text{A'}}{\text{A; B} \longrightarrow \text{A'; B}}$$

qui correspond à l'entrelacement à gauche de la composition parallèle. Donc le simple fait de supprimer l'entrelacement à droite permet de spécifier la séquence.

2. Codage des variables impératives

2.1. Le langage de décoration

2.1.1. DÉFINITION

On définit le langage de décoration $\mathcal{D}$ suivant comme une algèbre de termes avec variables avec les constructeurs suivants :

- **unify** : $\mathcal{D} \times \mathcal{D}$ pour la fonction ψ.
- **fork** : $\mathcal{D} \times \mathbf{N}$ qui servira à décorer l'opérateur $\|$.
- **succ** : $\mathcal{D} \times \mathbf{N}$ qui codera l'opérateur ';'.

– **call** : $\mathfrak{D}$ pour l'appel de fonction.

– $\top$ pour la décoration minimale.

– $\bot$ pour la décoration incohérente.

L'ensemble de termes considéré est la clôture de ces termes par l'opération associative commutative $\curlywedge$.

De plus on associe à chaque constructeur du langage de $\mathfrak{D}$ les opérations de même nom qui permettent de construire les termes :

On définit :

$$\begin{array}{rccc} \textbf{fork}: & \mathfrak{D} \times \mathbf{N} & \longrightarrow & \mathfrak{D} \\ & (t, n) & \longmapsto & \textbf{fork}(t, n) \end{array}$$

Exemple IV-3.: Opérateur **fork**

2.1.2. Ordonnancement des décorations

On définit l'ordre sur les décorations $\sqsubseteq$ comme la plus petite relation d'ordre qui vérifie les règles suivantes :

Règle 1 : $\forall n > 0, \forall t_i \in \mathfrak{D}, t_j \preceq \curlywedge_{i<n}(t_i)$.

Règle 2 : $\forall t_1, t_2 \in \mathfrak{D}, (t_1 \preceq \textbf{unify}(t_1, t_2)) \wedge (t_2 \preceq \textbf{unify}(t_1, t_2))$.

Règle 3 : $\forall t \in \mathfrak{D}, \forall i \in \mathbf{N}, t \preceq \textbf{fork}(t, i)$.

Règle 4 : $\forall t \in \mathfrak{D}, \forall i \in \mathbf{N}, t \preceq \textbf{succ}(t, i)$.

Règle 5 : $\forall t \in \mathfrak{D}, t \preceq \textbf{call}(t)$.

On aurait pu réécrire les quatre dernières règles par la règle suivante : Si t_1 est un sous-terme de t_2 alors $t_2 \sqsubseteq t_1$.

2.1.3. Précédence des décorations

De plus, le constructeur **succ** induit un pré-ordre $\prec_;$ sur les décorations de $\mathfrak{D}$. En effet des contraintes décorées par des décorations **succ**(t, i) avec i croissant correspondent à des contraintes séquentialisées. Donc, on dira qu'une décoration t_1 précédera une décoration t_2 si il existe deux entiers distincts i et j, et un sous-terme commun t_0 à t_1 et t_2 tels que $i < j$ et **succ**(t_0, i) soit un sous-terme de t_1 et **succ**(t_0, j) soit un sous-terme de t_2. On notera $t \# t'$ si $t \prec_; t' \prec_; t$ et $t \neq t'$.

On peut remarquer que $\prec_;$ est un ordre partiel et non un pré-ordre sur les décorations qui s'écrivent sans $\curlywedge$.

De plus, comme on l'a vu au début du chapitre, il sera toujours possible de plonger l'arbre d'exécution du programme dans un ordre total $\overline{\prec_;}$. Cet ordre est aussi un ordre total sur les décorations simples (voir après) de la base de faits puisque celles-ci sont posées par des agents du programme en question. On a un ordre et non un pré-ordre dans ce cas-là. Car la seule possibilité pour que deux décorations se précèdent l'une l'autre tout en étant disjointes est que l'une d'elle soit combinaison ($\sqcap$) de plusieurs sous-décorations, l'une précédant la deuxième décoration et l'autre la succédant.

2.1.4. Validité des décorations

Comme on le verra par la suite, la fonction de validation ζ est fonction de l'entrelacement des agents. C'est à partir du choix d'un ordre de succession des actions atomiques des agents (***ask*** et ***tell***) que l'on pourra spécifier les contraintes invalides et les contraintes valides, et ainsi restreindre la base de faits finale jusqu'à ce qu'elle corresponde à la base de faits finale qui serait fruit de l'exécution d'un programme `CC`étendu qui intégrerait la séquence et les variables impératives.

2.2. Décoration du code

On montre comment on trace l'exécution d'un programme `CC` avec la décoration. Cette étude se découpe en deux parties distinctes. La première traite de la manière dont on instrumente un code existant non décoré. La deuxième montre comment on peut programmer l'opération ';' et les variables impératives à l'aide de la décoration.

2.2.1. Instrumentation de code `CC`

Décoration initiale

La décoration initiale est $\bot$.

$$[[Init]]_{\bot}$$

Opérateur $\exists X$

Cette instruction n'a aucune influence sur la décoration courante puisqu'elle n'influe pas sur la trace d'exécution du programme.

$$[[\exists_X A]]_t \longrightarrow \exists_X [[A]]_t$$

Composition parallèle : $\|$

Comme dans le chapitre précédent, si A_i est la i-ème branche d'un opérateur de composition parallèle et si t est la décoration courante avant le $\|$, alors la décoration courante devient $\mathbf{fork}(t,\, i)$.

$$[[A_1 \| \ldots \| A_n]]_t \longrightarrow [[A_1]]_{\mathbf{fork}(t,1)} \| \ldots \| [[A_n]]_{\mathbf{fork}(t,n)}$$

*Opérateur **ask***

Pour instrumenter un ***ask***, on utilise le constructeur **unify** qui correspond à la fonction ψ de combinaison de décorations :

$$[[\mathbf{ask}(c) \rightarrow A]]_t \longrightarrow \exists_{t'} (\mathbf{ask}(\langle c, t' \rangle) \rightarrow [[A]]_{\mathbf{unify}(t,t')})$$

2.2.2. Implantation de l'opérateur ' ;'

Il s'implante en deux étapes. La première est une étape de parallélisation des agents séquentiels associée à une manipulation syntaxique de la décoration courante. Si A_i est la i-ème succession d'un opérateur de séquence ' ;' et si t est la décoration courante avant le ' ;', alors la décoration courante devient $\mathbf{succ}(t, i)$.

$$[[A_1; \ldots; A_n]]_t \longrightarrow [[A_1]]_{\mathbf{succ}(t,1)} \| \ldots \| [[A_n]]_{\mathbf{succ}(t,n)}$$

La deuxième étape est une opération de sélection a posteriori. Elle fait l'objet de la section suivante.

3. Contrôle d'exécution

Il s'agit donc d'examiner une base de faits finale et d'en supprimer les contraintes écrasées ainsi que leurs conséquences.

3.1. Choix d'un entrelacement

Le tri sur la base de faits se fera grâce à un entrelacement des agents en parallèle donné avant le tri. L'étude de l'influence de ce choix sur les écrasements se fera dans la section 3.3.2..

Le point de départ de ce choix est l'examen des décorations simples de la base de faits. Ces décorations montrent toutes les décorations courantes utilisées par des opérateurs *tell* lors de l'exécution du programme. A partir de ces décorations simples, il est facile de reconstruire un arbre d'exécution (en complétant bien sûr les sommets qui ne comportent pas de décorations).

Soit la base de faits du programme de l'exemple IV-2., il y a trois décorations simples dans la base de faits finale : $\mathbf{fork}(\mathbf{succ}(\top, 1), 1)$, $\mathbf{fork}(\mathbf{succ}(\top, 1), 2)$ et $\mathbf{succ}(\top, 2)$. A partir de là, il est facile de reconstruire l'arbre de la figure IV-1. En effet, on a les lexème $\mathbf{succ}(\top, 1)$ et $\mathbf{succ}(\top, 2)$, ce qui permet de voir qu'il y a deux parties du programme séparées par un ' ;'. Puis dans la première partie, on a deux agents en parallèle (lexème **fork**). L'arbre est reconstruit.

Exemple IV-4.: Reconstruction

La deuxième étape est le plongement de cette arbre de décoration dans un ordre total. Il y a deux manières de le faire. Soit on considère un entrelacement possible des opérateurs $\|$, soit on considère toutes les contraintes sur cet ordre impliquées par les conditions du traitement de l'exemple IV-2..

Le résultat est un ordre total ρ sur les décorations simples de la base de faits. Nous allons l'utiliser dans la section suivante pour séparer les contraintes licites et illicites.

3.2. Tri des contraintes

Soit ρ un entrelacement d'agents en parallèle donné, on propose un algorithme qui sépare les contraintes licites et les contraintes illicites. L'idée est de regarder les contraintes impliquées par le programme et de vérifier qu'elles sont bien issues des contraintes écrasables optimales.

Soit $\mathcal{B}$ une base de faits finale décorée par $\mathcal{D}$. Soit ρ un ordre total compatible avec $\mathcal{B}$ qui correspond à l'exécution d'un programme P qui a donné $\mathcal{B}$ comme base de faits finale. On se propose d'extraire une sous-base de faits $\zeta(\mathcal{B})$ de $\mathcal{B}$ qui corresponde à l'exécution de P, pour un ordre d'exécution donné par ρ, par un noyau `CC`étendu intégrant la séquence.

3.2.1. Découpage d'une base de faits finale

La première étape est un découpage de $\mathcal{B}$ en fonction des décorations. On ne s'intéresse déjà qu'à $\hat{\mathcal{B}}$, la partie atomique du point de vue de la décoration de $\mathcal{B}$.

De plus, on peut identifier $\rho(i)$ la décoration de la i-ème action du programme.

Puis on pose $\mathcal{B}_i$ le sous-ensemble de $\hat{\mathcal{B}}$ tel que toutes les décorations de $\mathcal{B}_i$, décorations de la forme $\diamond(t_1, \ldots, t_n)$, sont telles que $\forall j, t_j$ est inférieure à $\rho(i)$ et telles que l'un des t_j est égal à $\rho(i)$.

$\mathcal{B}_i$ est la frontière du sous-ensemble de $\hat{\mathcal{B}}$ dont les décorations sont inférieures à $\rho(i)$.

On peut remarquer que les $\mathcal{B}_i$ forment une suite finie souvent nulle, sauf quand $\rho(i)$ correspond à l'ajout d'une contrainte dans la base de faits.

3.2.2. Sélection des contraintes licites

Il reste maintenant à sélectionner dans chaque $\mathcal{B}_i$ la partie licite des contraintes décorées. Ceci se fait de manière récursive. $\mathcal{B}_0$ correspond à la base de faits initiale, qui est bien sûr licite. Soit $\zeta(\mathcal{B}_0) = \mathcal{B}_0$. On suppose maintenant que l'on a pu déterminer tous les $\zeta(\mathcal{B}_i)$ pour $i < n$, on détermine maintenant $\zeta(\mathcal{B}_n)$. Ce qui correspond à l'étape de récurrence.

D'abord, on définit ce que sont une décoration valide et une décoration invalide. Une décoration peut-être invalide pour plusieurs raisons :

Sous-terme invalide Si un sous-terme d'une décoration est invalide, alors la décoration est invalide. Réciproquement, une décoration est valide seulement si tous ses sous-termes sont valides.

Violation de la séquentialité Si un ***ask*** est activé par une contrainte qui lui succède (au sens de ρ) alors il y a violation de la séquentialité. Toute décoration qui montre ce symptôme, c'est-à-dire qui a un sous-terme de la forme $\mathbf{unify}(t, t')$ avec $t' \preceq_; t$, est invalide.

Décoration non optimale Seules les décorations correspondant à une contrainte optimale[2] peuvent être valides. Pour le système de contraintes introduit au début de ce chapitre, une contrainte générée est optimale si le **read** réagit avec la dernière instance de **write** qui le précède, ou avec le premier qui lui succède si aucun ne le précède.

Pour cela, on examine l'action du programme à « l'instant » $\rho(n)$. Soit $\langle c, \rho(n)\rangle$ la contrainte posée par le programme à cet instant. Toutes les contraintes de $\mathcal{B}_n$ ont donc été conséquences directes ou indirectes de $\langle c, \rho(n)\rangle$.

Tout d'abord, on examine $\rho(n)$. Si $\rho(n)$ est invalide, alors $\zeta(\mathcal{B}_n) = \emptyset$ et on a terminé notre étape de récurrence.

Sinon, on examine toutes les contraintes de $\hat{\mathcal{B}_n}$ et on élimine celles dont les décorations sont invalides d'après les critères précédents.

[2] C'est-à-dire conforme à la sémantique des variables impératives

Il ne reste plus qu'à poser $\zeta(\mathcal{B}_n) = \{$ l'ensemble des contraintes non invalides de $\hat{\mathcal{B}_n}$ $\}$ pour terminer l'étape de récurrence.

Une fois déterminée la suite des $\zeta(\mathcal{B}_n)$, il suffit de poser $\zeta(\mathcal{B}) = \bigcup_i \zeta(\mathcal{B}_i)$ pour définir la base de faits finale valide.

3.2.3. Validité de la sélection

Il s'agit ici de montrer deux choses, tout d'abord que la fonction ζ ainsi définie vérifie les axiomes d'une fonction de sélection. Deuxièmement, que la base de faits ainsi obtenue se conforme à la sémantique des variables impératives.

Validité de ζ

Cette première vérification est assez simple.

La compatibilité de la fonction ζ avec la fonction ψ est assurée par la règle dite de sous-termes invalides. Celle-ci assure que les deux membres d'un lexème **unify** doivent être valides.

La compatibilité de la fonction ζ avec l'ordre $\leqslant$ est une conséquence immédiate de la règle dite de sous-termes invalides et de la définition de l'ordre partiel $\leqslant$. Cela assure que si $t \leqslant t'$ et $\zeta(t) = \textbf{faux}$ alors $\zeta(t') = \textbf{faux}$.

Adéquation implantation-sémantique

Il faut montrer la complétude et la correction de l'implantation des variables impératives à l'aide de la décoration. Ces deux propriétés sont la conséquence des règles dites de violation de la séquentialité et de décoration non optimales.

La complétude (c'est-à-dire le faits que le programme décoré calcule tout ce que calcule un programme basé sur un CCétendu) est une conséquence de la règle d'optimalité des décorations. En effet, un noyau CCétendu n'utilisera que des contraintes optimales (puisqu'il intègre les spécifications des variables impératives). Il ne calculera pas plus. De plus, suite à l'utilisation de la règle de redémarrage, on est sûr que le programme CC décoré calculera au moins autant que le programme CCétendu.

Montrer la correction de la méthode revient à montrer le caractère suffisant des critères de validité des décorations. Ce qui se décrit en deux points. Le premier point correspond à la parallélisation des agents écrits en séquence

$$A_1; \ldots; A_n$$

Le seul cas visible de violation de cette séquence est la possibilité pour un ***ask*** contenu dans un agent A_i d'être activé par une contrainte posée par un agent $A_j, j > i$. Or ce cas est traité par la règle de violation de la séquentialité.

Le deuxième point consiste simplement à vérifier que le critère d'optimalité est bien complet. Ce qui va de soi si la règle n'a pas d'ambiguité. On vérifiera ce point pour le langage CC($\mathcal{M}$) dans le chapitre suivant.

3.3. Résultat

3.3.1. Reprise des exemples précédents

On reprend les deux exemples du début. On définit la relation d'équivalence $\simeq$ sur ce domaine de contrainte par $\mathbf{write}(X,\ \mathrm{a}) \simeq \mathbf{write}(Y,\ \mathrm{b}) \Leftrightarrow X = Y$. Les contraintes pivots sont les contraintes **read**.

Programme purement séquentiel

On peut maintenant instrumenter le code qui devient :

$$\begin{array}{l}\exists t(\\ \quad\quad \mathit{tell}(\mathbf{tag}(\mathbf{write}(X,0),\diamond(\mathbf{succ}(t,1))))\\ \quad \| \quad \mathit{tell}(\mathbf{tag}(\mathbf{write}(X,1),\diamond(\mathbf{succ}(t,1))))\\ \quad \| \quad \mathit{tell}(\mathbf{tag}(\mathbf{read}(Y,X),\diamond(\mathbf{succ}(t,3))))\\ \quad \| \quad \mathit{tell}(\mathbf{tag}(\mathbf{write}(X,2),\diamond(\mathbf{succ}(t,4))))\\ )\end{array}$$

Ces contraintes infèrent trois autres contraintes :

$$\begin{array}{l}\langle \mathbf{write}(Y,0),\diamond((\mathbf{succ}(t,3) \curlywedge \mathbf{succ}(t,1)))\rangle\\ \langle \mathbf{write}(Y,1),\diamond((\mathbf{succ}(t,3) \curlywedge \mathbf{succ}(t,2)))\rangle\\ \langle \mathbf{write}(Y,2),\diamond((\mathbf{succ}(t,3) \curlywedge \mathbf{succ}(t,4)))\rangle\end{array}$$

et d'après les règles de sélection, c'est bien la contrainte $\mathbf{write}(Y,\ 1)$ qui est privilégiée, car le **read** réagit avec le **write** optimal qui est celui juste antérieur.

Programme séquentiel-parallèle

Le programme séquentiel parallèle instrumenté est

$$\begin{array}{l}\exists t(\\ \quad\quad \mathit{tell}(\mathbf{tag}(\mathbf{write}(X,1),\diamond(\mathbf{fork}(\mathbf{succ}(t,1),1))))\\ \quad \| \quad \mathit{tell}(\mathbf{tag}(\mathbf{write}(X,2),\diamond(\mathbf{fork}(\mathbf{succ}(t,1),2))))\\ \quad \| \quad \mathit{tell}(\mathbf{tag}(\mathbf{read}(Y,X),\diamond(\mathbf{succ}(t,2))))\\ )\end{array}$$

qui génère les contraintes suivantes :

$$\begin{array}{l}\langle \mathbf{write}(Y,1),\diamond(\mathbf{succ}(t,2) \curlywedge \mathbf{fork}(\mathbf{succ}(t,1),1)\rangle\\ \langle \mathbf{write}(Y,2),\diamond(\mathbf{succ}(t,2) \curlywedge \mathbf{fork}(\mathbf{succ}(t,1),2)\rangle\end{array}$$

Et selon l'entrelacement des agents en parallèle, autour de la décoration $(\mathbf{succ}(t,\ 2)$, c'est soit la première contrainte, soit la deuxième qui écrase l'autre. On a donc deux résultats possibles : $\mathbf{write}(Y, 1)$ et $\mathbf{write}(Y, 2)$.

3.3.2. Influence du choix de l'entrelacement des agents

Dans cette section, on formalise l'influence du choix de l'entrelacement des agents sur le résultat final. Comme nous l'avons vu, la base finale sélectionnée est directement fonction du choix de l'entrelacement des agents. A partir de là, nous pouvons formaliser les concepts suivants :

Programme déterministe

Un programme sera dit *indépendant de l'ordre d'exécution* s'il est indépendant de l'entrelacement des agents. Une classe simple de ces programmes est celle des programmes déterministes qui n'utilisent pas d'écrasement de contraintes.

On peut affiner cette notion en introduisant une relation d'équivalence η sur les bases de faits finales. Un programme sera dit *indépendant de l'ordre d'exécution à une relation d'équivalence η près* si toute exécution du programme aboutit à une instance d'une même classe d'équivalence de la relation η. Une classe importante de programmes qui appartiennent à cette classification est celle des programmes qui utilisent le non-déterminisme du chapitre précédent sans utiliser l'écrasement de contraintes avec la fonction η qui identifie les solutions à un ordre près.

Parallélisation maximale

Si maintenant on se donne une base de faits finale, on peut se demander quels sont tous les entrelacements possibles des agents qui aboutissent à cette base de faits donnée. Si l'on considère dans ces entrelacements ceux qui donnent le bon résultat mais qui peuvent violer les contraintes de précédence dues à l'impérativité des variables, on obtient la classe de toutes les parallélisations possibles du programme.

Un exemple assez simple est celui du programme purement séquentiel du début du chapitre. Dans cet exemple,

$$\mathbf{write}(X,0);\mathbf{write}(X,1);\mathbf{write}(X,2);\mathbf{read}(Y,X);$$

On peut par exemple intervertir les deux premières contraintes **write** sans changer le résultat final.

4. Conclusion

Comme nous avons pu le voir, les variables impératives sont une forme restreinte de non-monotonie. En effet, elles n'impliquent pas le retrait des contraintes qui sont *écrasées* par le programme. Grâce à cette restriction importante, il est possible de rendre compte de cette extension par une sélection a posteriori sur la base de faits finale. Cette sélection est elle-même isomorphe à un contrôle sur l'exécution des agents et sur les résultats observables.

Finalement, à l'aide de la décoration et des extensions des CC qu'elle permet, il a été possible de définir clairement la sémantique du langage CC($\mathcal{M}$) que nous allons présenter dans le chapitre suivant.

Chapitre V

Présentation du langage CC($\mathcal{M}$)

Le langage CC($\mathcal{M}$) est basé sur un domaine de contraintes construit avec des messages entre objets. Il intègre en plus les constructeurs des CC déterministes de V. Saraswat [91] auxquels nous avons rajouté les deux extensions introduites dans les chapitres précédents. Le résultat est un langage objet parallèle muni du non-déterminisme. Si on lui adjoint un système de type adéquat et une implantation optimisée, alors le résultat est un langage noyau efficace et expressif pour implanter des langages de contraintes divers de manière générique.

1. CC($\mathcal{M}$), un langage objet

1.1. Un calcul de messages

1.1.1. Un langage sans unification

Comme nous avons pu le voir dans l'introduction, pour atteindre son objectif d'être un langage noyau pour la programmation concurrente par contraintes, le langage CC($\mathcal{M}$) doit pouvoir représenter des structures mutables.

Du point de vue de la représentation statique des structures de données, il est facile d'argumenter que les arbres à traits que l'on trouve dans Life[1], Oz [106] ou dans AKL [64] sont une extension utile des termes du premier ordre de la logique usuelle, en ce sens qu'ils offrent une meilleure lisibilité et permettent de déclarer plus facilement des objets incomplets.

Si l'on considère une personne comme étant un objet ayant deux attributs : l'âge et le nom ; quelqu'un dénommé John et âgé de 30 ans sera implanté de la manière suivante en Prolog :

$$PERSONNE(\text{"}John\text{"}, 30)$$

et de cette manière avec les arbres à traits :

$$PERSONNE(nom => \text{"}John\text{"}, age => 30)$$

Ce qui montre instantanément la différence de lisibilité. Maintenant, si l'on veut représenter toutes les personnes de 25 ans, cela s'écrira en Prolog,

$$\exists X, PERSONNE(X, 25)$$

et avec les arbres à trait :

$$PERSONNE(age => 25)$$

Et vérifier si une personne appartient à cette catégorie se fera en une étape d'unification dans Prolog et une étape de filtrage dans les arbres à traits.

Exemple V-1.: Expressivité des arbres à traits et des termes du premier ordre

---

Les deux paradigmes n'ont pas le même pouvoir d'expression, ils n'offrent pas les mêmes facilités. En particulier, les arbres à traits permettent de représenter des informations comme un terme et non plus comme un but (c'est la conclusion de l'exemple précédent où la structure incomplète : personne de 25 ans peut s'écrire comme un ψ-terme mais non pas comme un terme de Herbrand). Les ψ-termes offrent donc un gain d'expressivité, de lisibilité et de concision. Ces arguments sont développés dans [1].

Si l'on s'intéresse maintenant à l'évolution des informations et donc au caractère impératif des variables, on peut utiliser les travaux du chapitre précédent. Dans ce chapitre, nous avons montré comment, en brisant le caractère symétrique de l'unification en deux opérations distinctes : lecture et écriture, nous pouvons implanter dans un cadre cc des variables impératives et donc donner une sémantique simple de l'affectation destructive dans un contexte concurrent. C'est le même procédé qui est utilisé ici. Nous allons casser l'unification des arbres à traits en deux actions distinctes de lecture et d'écriture. Le résultat exposé dans la section suivante modélise exactement les messages entre objets.

1.1.2. Des messages de lecture et d'écriture

Comme pour les arbres à traits, nous utiliserons un ensemble dénombrable $\mathfrak{A}$ d'attributs, dénotés par des identificateurs : *âge*, *nom*. De même, nous utiliserons un ensemble infini dénombrable $\mathcal{V}$ de variables dénotées X ou Y. Enfin, on suppose donné un ensemble dénombrable $\mathcal{B}$ de valeurs simples.

A partir de là, nous pouvons donner les constructeurs du domaine de messages. Chaque constructeur (lecture ou écriture) se déclinera en deux versions suivant qu'il s'adresse à un objet ou au champ d'un objet.

On trouve donc les constructeurs suivants :

write$(X,\ v)$ pour $X \in \mathcal{V}$ et $v \in \mathcal{B}$.
write$(X,\ Y)$ pour $X, Y \in \mathcal{V}$.
read$(X,\ Y)$ pour $X, Y \in \mathcal{V}$.

et leurs contreparties portant sur les attributs des variables :

write$_{\boldsymbol{a}}(X, v)$ pour $X \in \mathcal{V}$, $\boldsymbol{a} \in \mathfrak{A}$ et $v \in \mathcal{B}$.
write$_{\boldsymbol{a}}(X,\ Y)$ pour $X, Y \in \mathcal{V}$ et $\boldsymbol{a} \in \mathfrak{A}$.
read$_{\boldsymbol{a}}(X,\ Y)$ pour $X, Y \in \mathcal{V}$ et $\boldsymbol{a} \in \mathfrak{A}$.

La signification de ces atomes est simple. L'atome **write**$(X,\ v)$ signifie que la valeur v est écrite sur la variable X. L'atome **write**$(X,\ Y)$ signifie lui que la variable Y considérée

comme une valeur est écrite sur la variable X. Enfin, $\mathbf{read}(X, Y)$ signifie que le contenu de la variable X est écrit sur la variable Y. Il en est de même pour les constructeurs avec attribut. $\mathbf{write}_{\boldsymbol{a}}(X, v)$ signifie que la valeur v est écrite sur le champ $\boldsymbol{a}$ de la variable X. Pour finir, $\mathbf{read}_{\boldsymbol{a}}(X, Y)$ signifie que le contenu du champ $\boldsymbol{a}$ de la variable X est écrit sur la variable Y.

Il ne reste plus maintenant qu'à expliciter les interactions entre ces différents atomes. L'idée directrice est une simple notion de lecture-écriture dans une case mémoire. Ainsi peut-on donner les règles suivantes :

$$\mathbf{write}(X, v), \mathbf{read}(X, Y) \vdash \mathbf{write}(Y, v)$$

$$\mathbf{write}(X, Z), \mathbf{read}(X, Y) \vdash \mathbf{write}(Y, Z)$$

pour le cas sans attribut,

$$\mathbf{write}_{\boldsymbol{a}}(X, v), \mathbf{read}_{\boldsymbol{a}}(X, Y) \vdash \mathbf{write}(Y, v)$$

$$\mathbf{write}_{\boldsymbol{a}}(X, Z), \mathbf{read}_{\boldsymbol{a}}(X, Y) \vdash \mathbf{write}(Y, Z)$$

pour le cas avec attribut.

Il ne reste plus qu'un seul lot de règles pour traduire le transport des attributs d'une variable vers une autre :

$$\mathbf{write}(X, Y), \mathbf{write}_{\boldsymbol{a}}(Y, v), \mathbf{read}_{\boldsymbol{a}}(X, Z) \vdash \mathbf{write}(Z, v)$$

$$\mathbf{write}(X, Y), \mathbf{write}_{\boldsymbol{a}}(Y, U), \mathbf{read}_{\boldsymbol{a}}(X, Z) \vdash \mathbf{write}(Z, U)$$

REMARQUE V-2. ($\mathcal{M}$ ET LES ψ-TERMES)
Les messages de $\mathcal{M}$ implantent une variante sans unification des ψ-termes [64] ou de leurs variantes (Algèbre OSF, Enregistrements [106], arbres à trait [66]). Si les deux paradigmes reposent sur les même bases : ensemble d'attributs, forme d'arbres, le résultat est complètement différent. Par rapport à un système avec unification, le système à base de lecture-écriture est *asymétrique* : $\mathbf{read}(X, Y)$ implique que si X a une valeur, Y a la même mais pas l'inverse, en fait $X = Y$ se rapproche de $\mathbf{read}(X, Y) \wedge \mathbf{read}(Y, X)$; *asynchrone* :$\mathbf{read}(X, Y)$ n'arrête pas le calcul ; *Sans erreur* : Si $\mathbf{write}(X, a)$ et $\mathbf{write}(Y, b)$ alors $\mathbf{read}(X, Y)$ implique $\mathbf{write}(Y, a)$ et non pas une erreur d'unification.

1.1.3. PROPRIÉTÉS DU DOMAINE DE MESSAGES

Le domaine des messages de la section précédente est un ensemble d'objets muni d'une relation de conséquence logique monotone. C'est donc un système de contraintes. Ce qu'exprime le lemme suivant :

LEMME V-3. (SYSTÈME DE CONTRAINTE $\mathcal{M}$)
Soit $\mathcal{M}$ l'ensemble des messages précédemment introduits muni de la règle de conséquence logique exprimée par les règles précédentes en syntaxe SOS. C'est donc un système de contraintes au sens des cc.

1.2. Les instructions du langage CC($\mathcal{M}$)

1.2.1. Les déclarations

Il y a quatre sortes de déclarations, les déclarations de fonctions, d'objets, de classes et de variables locales. Ces déclarations forment le contenu d'un module. Les déclarations de classes, d'objets ou de variables sont implantées au dessus du CC($\mathcal{M}$) par une création ($\exists$) d'une variable de $\mathcal{V}$. Les méthodes (qui peuvent être surchargées) sont implantées à la compilation en les regroupant par nom derrière une fonction unique qui sélectionne la bonne méthode à l'exécution. Mais le compilateur peut toujours faire cette sélection à la compilation s'il a assez d'informations.

Modules

Les modules sont des espaces lexicaux où des fonctions, des classes et des objets sont déclarés. Il s'écrivent comme suit :

```
Prog = 'MODULE' Ident
'IMPORT' IdentList 'END'
'EXPORT' IdentList 'END'
Declarations .
```

où `Ident` est le nom du module et les `IdentList` sont la liste de modules importés et la liste des noms (fonctions, classes, objets) exportés.

Les déclarations sont de plusieurs types :

Déclaration de classe

Dans le langage CC($\mathcal{M}$), on crée une classe en posant :

```
NouvelleClasse <: TermeDeClasse
```

où `TermeDeClasse` à la forme suivante :

```
TermeDeClasse  =  Ident [ FeatureTypes ] .
FeatureType = <HIDDEN> Ident ':' Type .
```

Le mot clef optionnel `HIDDEN` sert juste à protéger les champs en les cachant.

On peut définir la classe *Personne* en posant :

```
Personne < : Objet [ nom : Word, age : Integer ]
```

Exemple V-4.: Création de classe

Déclaration d'objets

On déclare un objet avec la syntaxe suivante :

```
NouvelObjet :: Terme
```

où un `Terme` à la forme suivante :

```
Terme       ::= Ident
              |  Ident [ FeatureValues ]


FeatureValue ::= Ident <- Arg .
```

On peut instancier la classe des Personnes en créant l'objet suivant :

```
JOHN : : Personne [ age <- 30, nom <- "John" ]
```

Exemple V-5.: Création d'objets

Déclaration de variable globale

Pour déclarer une variable globale, il suffit d'écrire le code suivant :

```
VAR NomDeVariable : Type
```

où un `Type` est un identificateur qui représente une classe ou un type système (*entier*, *booléen*, *mot* ou *réel*).

Déclaration de fonction

On implante une nouvelle fonction en écrivant :

```
FUN NomDeFonction ( Arguments ) : Type :=
  VariablesLocales
  Codes
END
```

où les arguments (s'il y en a) sont écrits de la manière suivante :

```
NomDeChamp : Type
```

où les variables locales sont introduites comme cela

```
VAR NomDeVariable : Type
```

et où le `Code` est une suite d'instructions du langage CC($\mathcal{M}$). Ces instructions seront présentées dans la section suivante.

On peut aussi importer en CC($\mathcal{M}$) des fonctions C en posant :

```
FUN NomDeFonction ( Arguments ) : Type := C ( "NomDeFonctionExterne" )
```

1.2.2. Les instructions de contrôle

Celles-ci sont de deux types, avec ou sans itération.

Sans itération, c'est le `IF .. THEN .. ELSE`, qui est enrichi par une expression `ELSEIF`, une simple contraction syntaxique pour `ELSE IF`.

Une instruction de ce type s'implante facilement dans le `CC` avec séquence. En effet `IF Test Then I1 Else I2` devient :

```
EvalueTest(VariableTemporaire) ;
(
  ask(write(VariableTemporaire, true))  -> I1
||
  ask(write(VariableTemporaire, false)) -> I2
)
```

Avec itération, on trouve le `REPEAT ... UNTIL ...` et `WHILE ... DO ... END` suivant que l'on effectue ou pas le corps de la boucle avant le premier test de fin.

De la même manière que pour le `IF THEN ELSE`, le `REPEAT UNTIL` s'implante au dessus de `CC` avec séquence grâce à une fonction récursive auxiliaire. `Repeat I Until Test` devient :

```
RepeatAux := I ;
  EvalueTest(VariableTemporaire) ;
  ask(write(VariableTemporaire, true))  -> RepeatAux
End
```

avec dans le code, à la place du `Repeat Until`, le simple appel à la fonction `RepeatAux()`.

1.2.3. Les Instructions de lecture-écriture

La conversion la plus facile entre CC($\mathcal{M}$) et CC $+\mathcal{M}$ se trouve dans les instructions de lecture-écriture. En effet, pour poser

```
X <- Expression
```

il suffit d'écrire :

```
EvalueExpression(VariableRetour); write(X, VariableRetour)
```

De même, pour lire le champ a de la variable Y, ce que l'on notera (`a(Y)`), il suffit d'écrire :

```
Read_a(Y, ValeurLue)
```

et d'utiliser `ValeurLue` dans la suite du calcul.

1.2.4. La gestion du parallélisme

La gestion des processus est directement inspirée de la définition des CC. En effet, on trouve dans le langage CC($\mathcal{M}$) les constructeurs suivants qui sont directement tirés sans conversion des CC.

- L'instruction `DETACH f()` implante la composition parallèle ($\|$). Elle crée un nouvel agent chargé d'exécuter la procédure `f()`.
- L'instruction `DETACH local f()` implante la disjonction angélique d'agents ($\oplus$). Elle crée un nouvel agent chargé d'exécuter la procédure `f()` avec un segment de mémoire local (voir section 3.2.).
- La séquence (';') est implantée directement dans le langage sans syntaxe particulière. Deux instructions qui se suivent dans un bloc s'exécutent de manière séquentielle.

1.2.5. La gestion des exceptions

Grâce aux mécanismes du parallélisme, il est possible d'implanter dans CC($\mathcal{M}$) une gestion d'exceptions.

En effet, on peut écrire

```
Y <- Catch f()
```

où `Catch` récupère les exceptions levées par un `Raise E`. `Y` prend alors la valeur de l'exception levée ou `Nil` si aucune n'est levée (ce qui est implanté par un `Raise Nil` à la fin de la procédure `f()`). Cette instruction `Catch` est implantée comme suit :

```
Exists Tampon
(
```

```
  f(Tampon)
||
  read(Tampon, Y)
)
```

et le `raise E` est juste une écriture sur `Tampon` suivi d'un arrêt de l'agent :

```
write(Tampon, E) ; stop
```

Donc, grâce au mécanisme de synchronisation lecture-écriture de $\mathcal{M}$, le `Catch` est évalué dès que l'exception est levée, ce qui est le résultat attendu.

2. Le typage dans CC($\mathcal{M}$)

2.1. Le typage orienté-objet

2.1.1. Le compromis typage fort-expressivité

Le typage a deux fonctions principales. La première fonction est une fonction de vérification : il s'agit d'éliminer les messages non définis, c'est-à-dire la lecture ou l'écriture sur des champs non définis, ou encore l'appel de méthode inexistante. La deuxième fonction est une fonction d'optimisation. En effet, plusieurs objets peuvent avoir chacun une méthode ayant le même nom. Il y a donc surcharge des noms de fonctions et il faut décider à un moment ou à un autre quelle méthode effectivement employer. C'est là qu'intervient le système de type car si à la compilation, on peut s'assurer de l'unicité de la méthode à appeler, on peut alors remplacer l'appel fonctionnel générique par un appel direct à la bonne méthode. On parle alors d'attachement statique. C'est une optimisation importante dans les langages objets car l'appel fonctionnel générique est très coûteux. Mais le typage peut permettre une deuxième optimisation, car si la nature des valeurs qui passent dans les messages (entier, booléen, réel en double précision) est connue à la compilation, alors on peut simplifier l'implantation de ces valeurs et effectuer une « sortie de boîte » (en anglais *unboxing*) qui supprime la boite qui entoure chaque valeur, boite qui se compose en général de la valeur de l'objet ainsi que d'une information sur sa nature. On trouve là les deux optimisations principales des langages objets.

2.1.2. Le choix d'un double langage de type

Malheureusement, on ne peut combiner typage statique, optimisation et puissance d'expression. Soit on privilégie le typage et l'efficacité au dépens de l'expressivité, soit l'inverse.

Le langage CC($\mathcal{M}$) présente un compromis original pour le typage. Au lieu de se contenter du seul langage de type issu des classes ($\mathcal{T}^1$), on en a considéré deux ($\mathcal{T}^0$ et $\mathcal{T}^1$). Le langage de type $\mathcal{T}^0$ représente la nature des valeurs (entier, booléen, objet, mot). Le langage de type $\mathcal{T}^1$ implante une version raffinée de $\mathcal{T}^0$ auquel on aurait rajouté la hiérarchie de classe. Ces deux langages étant d'expressivités différentes, on peut leur imposer des conditions différentes. En particulier, le typage associé à $\mathcal{T}^0$ sera fort et statique et celui associé à $\mathcal{T}^1$ sera dynamique et pas fort.

2.2. Les deux langages de type

Nous allons présenter le typage de $\mathcal{M}$ comme une extension de $\mathcal{M}$. En effet, pour typer les contraintes de lecture et d'écriture, nous avons ajouté à $\mathcal{M}$ de nouvelles contraintes véhiculant des informations de type.

Dans une telle présentation, l'inférence de type est équivalente à la conséquence logique. Le calcul de type statique ou dynamique est donc une simple propagation de contraintes.

De plus, étant donné que les types deviennent des objets manipulables par le programme, il devient possible d'implanter la surcharge des méthodes grâce à cette extension.

Si la fonction `f` possède deux méthodes `f1` et `f2`, et un argument `X`, on peut implanter la sélection dynamique de la méthode à employer comme un test sur le type de `X`.

```
FUN F(X) :TypeDeF :=
If Type(X) = Type1 Then f1(X) Elseif Type(X) = Type2 Then f2(X)
Else raise ErreurDeType.
End
```

Exemple V-6.: Sélection d'une méthode

2.2.1. $\mathcal{T}^0$ ET LA REPRÉSENTATION INTERNE DES VALEURS

Les atomes de $\mathcal{T}^0$

Comme nous l'avons vu précédemment, le langage de type $\mathcal{T}^0$ sert à décrire la représentation interne des variables et des champs des objets. On rajoute donc des atomes à $\mathcal{M}$ pour représenter des informations de type au niveaux des variables :

- $\mathbf{isinteger}^0(X)$ pour $X \in \mathcal{V}$.
- $\mathbf{isboolean}^0(X)$ pour $X \in \mathcal{V}$.
- $\mathbf{isreal}^0(X)$ pour $X \in \mathcal{V}$.
- $\mathbf{isword}^0(X)$ pour $X \in \mathcal{V}$.
- $\mathbf{isobject}^0(X)$ pour $X \in \mathcal{V}$.
- $\mathbf{isinteger}^0_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\mathbf{isboolean}^0_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\mathbf{isreal}^0_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\mathbf{isword}^0_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\mathbf{isobject}^0_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.

Comme les informations de typage sont disjointes, on peut séparer les variables en deux camps disjoints. Celles typées par $\mathbf{isinteger}^0$, $\mathbf{isboolean}^0$, $\mathbf{isreal}^0$, $\mathbf{isword}^0$ accepteront une valeur élément de $\mathcal{B}$ (c'est-à-dire un atome de la forme $\mathbf{write}(X, v)$ où $v \in \mathcal{B}$). Les autres typées par $\mathbf{isobject}^0$ accepteront des valeurs dans $\mathcal{V}$ (c'est-à-dire des atomes de la forme $\mathbf{write}(X, Y)$ où $X, Y \in \mathcal{V}$). Et cette disjonction est totale. Aucune variable typée de la première manière n'acceptera des valeurs de type $\mathbf{isobject}$ et réciproquement.

Caractère fortement typé de $\mathcal{T}^0$

On impose au langage CC($\mathcal{M}$) le fait que le typage relatif à $\mathcal{T}^0$ soit fort. Ceci peut être simplement décrit par la règle opérationnelle non monotone suivante :

$$\begin{aligned} \textbf{write}_{\boldsymbol{a}}(X, Z), \neg(\textbf{isobject}_{\boldsymbol{a}}(X)) &\vdash fail \\ \textbf{write}_{\boldsymbol{a}}(X, v), \neg(\textbf{is}\langle\langle\ \textbf{type}\ \rangle\rangle_{\boldsymbol{a}}(X) \mid v \in \langle\langle\ \textbf{type}\ \rangle\rangle) &\vdash fail \end{aligned}$$

où « **type** » représente **integer**, **boolean**, **real** ou **word**. On obtient bien ce que l'on veut, c'est-à-dire que toute expression mal typée ou non typée génère une erreur.

2.2.2. Le langage de type des classes $\mathcal{T}^1$

Le langage de type $\mathcal{T}_1$ est plus raffiné que le langage $\mathcal{T}_0$. Il est utilisé dans la définition des domaines de fonctions et des classes. Il est nécessaire pour la surcharge des fonctions.

Codage du typage $\mathcal{T}_1$ dans $\mathcal{M}$

Il se présente comme une version plus complexe du langage $\mathcal{T}^0$. Tout d'abord il reprend certains des atomes du langage $\mathcal{T}^0$:

- $\textbf{isinteger}^1(X)$ pour $X \in \mathcal{V}$.
- $\textbf{isboolean}^1(X)$ pour $X \in \mathcal{V}$.
- $\textbf{isreal}^1(X)$ pour $X \in \mathcal{V}$.
- $\textbf{isword}^1(X)$ pour $X \in \mathcal{V}$.
- $\textbf{isinteger}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{isboolean}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{isreal}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{isword}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.

Puis il complique les atomes **isobject** en spécifiant quelle variable porte les informations de type.

- $\textbf{isobjecttype}^1(X, Y)$ pour $X \in \mathcal{V}, Y \in \mathcal{V}$.
- $\textbf{isobjecttype}^1_{\boldsymbol{a}}(X, Y)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}, Y \in \mathcal{V}$.

Et ces variables « porteuses » que nous appellerons plus tard des classes sont liées à une autre famille d'atomes de typage :

- $\textbf{hasinteger}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{hasboolean}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{hasreal}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{hasword}^1_{\boldsymbol{a}}(X)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}$.
- $\textbf{hasobjecttype}^1_{\boldsymbol{a}}(X, Y)$ pour $X \in \mathcal{V}, \boldsymbol{a} \in \mathcal{A}, Y \in \mathcal{V}$.

Lien entre $\mathcal{T}^0$ et $\mathcal{T}^1$

On peut maintenant facilement lier les deux langages de type. Il est clair que $\mathcal{T}^1$ implique $\mathcal{T}^0$. Les règles sont les mêmes, mis à part un affaiblissement de l'atome **isobjecttype** en **isobject**.

$$\begin{aligned} \textbf{isinteger}^1(X) &\vdash \textbf{isinteger}^0(X) \\ \textbf{isboolean}^1(X) &\vdash \textbf{isboolean}^0(X) \\ \textbf{isreal}^1(X) &\vdash \textbf{isreal}^0(X) \\ \textbf{isword}^1(X) &\vdash \textbf{isword}^0(X) \\ \textbf{isobjecttype}^1(X, Y) &\vdash \textbf{isobject}^0(X) \end{aligned}$$

et de même pour les attributs d'une variable :

$$\begin{aligned}
\mathbf{isinteger}^1_{\boldsymbol{a}}(X) &\vdash \mathbf{isinteger}^0_{\boldsymbol{a}}(X)\\
\mathbf{isboolean}^1_{\boldsymbol{a}}(X) &\vdash \mathbf{isboolean}^0_{\boldsymbol{a}}(X)\\
\mathbf{isreal}^1_{\boldsymbol{a}}(X) &\vdash \mathbf{isreal}^0_{\boldsymbol{a}}(X)\\
\mathbf{isword}^1_{\boldsymbol{a}}(X) &\vdash \mathbf{isword}^0_{\boldsymbol{a}}(X)\\
\mathbf{isobjecttype}^1_{\boldsymbol{a}}(X,Y) &\vdash \mathbf{isobject}^0_{\boldsymbol{a}}(X)
\end{aligned}$$

Grâce à ces conversions automatiques, on va pouvoir identifier les atomes homonymes des deux langages et oublier les exposants ^{1} et ^{0}.

Spécification des règles de typage

Par cette conversion entre $\mathcal{T}^1$ et $\mathcal{T}^0$, il n'y a pas lieu de spécifier des règles de typage entre les atomes d'écriture (**write**) et les atomes de typage du $\mathcal{T}^1$.

Maintenant on va utiliser un attribut particulier ***is_a***. Grâce à cet attribut, on va pouvoir lier une classe à la classe porteuse. Ceci est fait de la manière la plus simple en déclarant $\mathbf{write}_{\boldsymbol{is\_a}}(X,Y)$ qui spécifie que Y est la classe de X.
A partir de là, on peut lier le type d'un objet à sa classe :

$$\mathbf{write}_{\boldsymbol{is\_a}}(X,Y) \vdash \mathbf{isobjecttype}^1(X,Y)$$

Cependant, si Y est la classe de X, alors toutes les informations de type portées par Y (atomes de la forme **hasinteger**) se transportent sur X :

$$\begin{aligned}
\mathbf{write}_{\boldsymbol{is_a}}(X,Y), \mathbf{hasinteger}^1_{\boldsymbol{a}}(Y) &\vdash \mathbf{isinteger}_{\boldsymbol{a}}(X)\\
\mathbf{write}_{\boldsymbol{is_a}}(X,Y), \mathbf{hasboolean}^1_{\boldsymbol{a}}(Y) &\vdash \mathbf{isboolean}_{\boldsymbol{a}}(X)\\
\mathbf{write}_{\boldsymbol{is_a}}(X,Y), \mathbf{hasreal}^1_{\boldsymbol{a}}(Y) &\vdash \mathbf{isreal}_{\boldsymbol{a}}(X)\\
\mathbf{write}_{\boldsymbol{is_a}}(X,Y), \mathbf{hasword}^1_{\boldsymbol{a}}(Y) &\vdash \mathbf{isword}_{\boldsymbol{a}}(X)\\
\mathbf{write}_{\boldsymbol{is_a}}(X,Y), \mathbf{hasobjecttype}^1_{\boldsymbol{a}}(Y,Z) &\vdash \mathbf{isobjecttype}_{\boldsymbol{a}}(X,Z)
\end{aligned}$$

Ces règles traduisent simplement le fait que l'objet X est une instance de la classe Y : tous les types d'attributs attachés à la classe Y se transportent sur l'objet X.

2.2.3. Les classes et l'héritage

Maintenant que ces deux langages de type sont définis, on peut aller plus loin dans le typage de $\mathcal{M}$ et définir une notion de classe et d'héritage.

Les classes

On recode les classes et les objets dans le langage de $\mathcal{M}$. Les objets sont des enregistrements extensibles [12]. Avec l'attribut ***superclass***, on implante de manière circulaire la méta-classe $CLASSE$ par le programme suivant :

$\exists\, CLASSE \quad ($
$\quad \mathbf{write}_{\mathit{is\_a}}(CLASSE,\, CLASSE)$
$\quad \mathbf{hasobjecttype}^1_{\mathit{is\_a}}(CLASSE,\, CLASSE)$
$\quad \mathbf{write}_{\mathit{superclass}}(CLASSE,\, \mathit{object})$
$\quad \mathbf{hasobjecttype}^1_{\mathit{superclass}}(CLASSE,\, CLASSE)$
$)$

Les variables de type objet sont séparées en deux camps disjoints : celles de type $CLASSE$ et les autres. Celles de type $CLASSE$ sont liées entre elles par l'attribut ***superclass***. Elles forment un arbre dont le sommet est la classe *objet* définie par le programme suivant :

$\exists\, \mathit{objet} \quad ($
$\quad \mathbf{write}_{\mathit{is\_a}}(\mathit{objet},\, CLASSE)$
$\quad \mathbf{write}_{\mathit{superclass}}(\mathit{objet},\, \mathit{objet})$
$)$

objet est donc sa propre super-classe. Maintenant soit l'arbre $\mathcal{C}T$ des classes défini de la manière suivante :X est un fils de Y si et seulement si $\mathbf{write}_{\mathit{superclass}}(Y, X)$. On demande à cet arbre d'être un arbre enraciné, c'est-à-dire sans boucles et sans points isolés. On est donc retombé sur un arbre de classe avec un héritage simple.

L'héritage

Il ne reste plus qu'à définir la notion d'héritage entre ces classes. Si une classe Y hérite d'une classe X, alors elle récupère toutes les informations de type de cette dernière :

$$\begin{aligned}
\mathbf{write}_{\mathit{superclass}}(Y, X), \mathbf{hasinteger}^1_a(X) &\vdash \mathbf{hasinteger}_a(Y) \\
\mathbf{write}_{\mathit{superclass}}(Y, X), \mathbf{hasboolean}^1_a(X) &\vdash \mathbf{hasboolean}_a(Y) \\
\mathbf{write}_{\mathit{superclass}}(Y, X), \mathbf{hasreal}^1_a(X) &\vdash \mathbf{hasreal}_a(Y) \\
\mathbf{write}_{\mathit{superclass}}(Y, X), \mathbf{hasword}^1_a(X) &\vdash \mathbf{hasword}_a(Y) \\
\mathbf{write}_{\mathit{superclass}}(Y, X), \mathbf{hasobjecttype}^1_a(X, Z) &\vdash \mathbf{hasobjecttype}_a(Y, Z)
\end{aligned}$$

Il ne reste plus maintenant qu'à considérer le cas où lors de l'héritage, le programme redéfinit le typage d'un attribut. En d'autres termes, que se passe t-il quand il y a conflit entre deux atomes de type **has**« **type** » ?

On a deux possibilités. Soit les types sont incompatibles (entre un **hasinteger** et un **hasreal** ou entre un **hasinteger** et un **hasobjecttype**), dans ce cas le résultat est $\top$, c'est-à-dire **faux**. Soit on a deux atomes $\mathbf{hasobjecttype}_a(X, Y)$ et $\mathbf{hasobjecttype}_a(X, Z)$, alors on regarde le lien qui existe entre X et Y dans $\mathcal{C}T$: si l'une des deux variables est un descendant de l'autre, c'est elle qui impose le typage, sinon, il y a incohérence et ces deux atomes impliquent **faux**.

2.3. La vérification de type dans CC($\mathcal{M}$)

La vérification de type dans le langage CC($\mathcal{M}$) couvre deux aspects : la définition et l'appel de fonctions ; la lecture et l'écriture sur les objets. Pour vérifier le type de ces opérations, on applique des règles simples.

2.3.1. Les fonctions

Le typage des valeurs retours des fonctions est effectué de manière statique d'après leurs définitions. On impose que chaque méthode d'une même fonction soit homogène selon $\mathcal{T}^0$.

De plus, on typera chaque appel de fonctions comme le plus petit type majorant (au sens de $\mathcal{T}^1$) les type de retour des toutes les restrictions compatibles avec les types des arguments. C'est une manière classique de typer les appels fonctionnels avec des multi-méthodes [22, 19, 20].

2.3.2. Lecture, écriture

Il s'agit maintenant de vérifier les messages de lecture et d'écriture. Pour cela, on compare le type des deux arguments des atomes **write** et **read**.

La vérification de type est simple. Soient les atomes suivants :

$$\mathbf{write}(X, v), \mathbf{istype1}(X), \mathbf{read}(X, Y), \mathbf{istype2}(Y)$$

alors pour le langage $\mathcal{T}^0$:

- l'atome **write** est bien typé si le type de v est égal à celui de X (**type1**).
- l'atome **read** est bien typé si **type1** et **type2** sont égaux.

Pour le langage $\mathcal{T}^1$:

- l'atome **write** est bien typé si le type de v est inférieur à celui de X (**type1**).
- On a trois cas pour l'atome **read** :
  1. **type1** $\leqslant$ **type2** alors l'atome **read** est bien typé car v est de type **type2** puisqu'il est de type **type1**.
  2. **type2** $\leqslant$ **type1**, il peut y avoir un mauvais typage à l'exécution (ce qui est une conséquence directe du fait que le typage associé à $\mathcal{T}^1$ n'est pas fort). L'atome **read** est donc potentiellement mal typé. Le langage doit donc implanter un test de type dynamique et lever une exception s'il y a violation du typage. (v n'est pas de type **type2**).
  3. **type1** et **type2** sont incomparables, l'instruction est à coup sûr mal typée. Il y a donc une erreur de typage. Elle est détectée à la compilation.

2.4. Conclusion sur le typage

L'implantation de CC($\mathcal{M}$) impose que le typage associé à $\mathcal{T}^0$ soit fort. Le langage CC($\mathcal{M}$) qui en résulte reste suffisamment expressif pour nos besoins (le polymorphisme d'objets et la surcharge des fonctions permettent d'implanter des algorithmes génériques). De plus, il peut être optimisé à la compilation de deux manières. Tout d'abord, comme on connaît la représentation interne de chaque valeur, leur implantation se fait grâce à un type simple du langage C. Enfin, le seul typage dynamique se fait sur les objets, or ceux-ci portent en eux la classe dont ils sont l'instance. Ceci couplé au fait que l'héritage est simple nous a permis d'implanter un typage dynamique (pour l'appel des fonctions surchargées et pour la

vérification de atomes **read**) efficace, d'autant plus qu'il s'applique rarement[1]. Ces résultats argumentent en faveur du choix du double langage de type de CC($\mathcal{M}$).

3. Un modèle d'exécution de CC($\mathcal{M}$)

3.1. Un modèle d'exécution pour un CC déterministe

Le langage CC($\mathcal{M}$) ainsi présenté est donc un CC déterministe basé sur un système de contraintes décoré de messages entre objets. Il bénéficie à ce titre d'un modèle d'exécution très simple du fait de la simplicité même des constructeurs d'un CC déterministe. Outre la gestion des constructeurs génériques (quantification, composition parallèle, appel de procédure), il convient de s'intéresser aux cas de suspensions d'un agent et aux interactions entre la base de faits et les agents.

3.1.1. Les règles de suspension

Les règles qui gèrent les interactions entre les agents et la base de faits sont simples. Elles sont de deux natures, qui correspondent à l'ajout ou à la demande d'une contrainte. Pour simplifier la présentation de ces interactions, et sans perdre la généralité du propos, nous nous intéresserons au système de contraintes simplifié présenté dans la section 1.2. du chapitre IV.

Cas de la demande d'une contrainte

Lorsqu'un agent demande une certaine contrainte, celle-ci est soit un atome **write**, soit un atome **read**. En pratique, seuls les atomes de type **write** sont demandés par le langage CC($\mathcal{M}$). Dans ce cas, l'agent se suspend seulement s'il n'y a pas de contrainte **write** portant sur la bonne variable, ou si celle-ci n'a pas la bonne valeur.

En pratique, on rencontrera deux types de demandes, les premières concerneront l'existence d'un atome **write** indépendamment de sa valeur. On les trouvera lors de l'appel de procédures pour vérifier que tous les paramètres sont bien instanciés. Ce type de demande est équivalent au Freeze[100] de Prolog. Il peut être implanté simplement.

Le deuxième type de demande suppose une valeur précise attendue. Ce type de demande intervient dans les tests booléens lors des instructions `If`, `Repeat` et `While`. Ce type de demande est plus complexe à gérer que le premier type, puisque l'ajout d'un atome **write** avec la mauvaise valeur induit un test mais pas le réveil de l'agent.

Cas de l'ajout d'une contrainte

Lorsqu'un agent ajoute un atome **write** ou un atome **read**, celui-ci peut générer d'autres atomes **write** directement. C'est à ce moment-là que l'on doit vérifier s'il n'y a pas un ou plusieurs agents suspendus dans l'attente de cette contrainte. Dans ce cas-là, ces agents doivent être réactivés.

Calcul asynchrone

Si on regarde bien, la symétrie brisée induite par l'atome **read** par opposition à la contrainte $X = Y$ transforme le calcul synchrone de lecture-écriture en un calcul asynchrone.

[1] Le module de compilation du langage CC($\mathcal{M}$) ne comporte qu'une seule fonction surchargée, c'est la fonction récursive d'auto-compilation d'une instruction. Le module de typage n'en contient lui aussi qu'une, celle qui demande à une instruction de se typer elle même.

En effet, la lecture d'une variable par une contrainte **read** ne se suspend pas si cette variable n'est pas instanciée. Elle n'est pas blocante. Le calcul peut continuer. Pour synchroniser sur l'écriture, il faut explicitement demander avec un ***ask***.

Finalement, si ce n'était pour les instructions de contrôle (qui sont bien sûr fondamentales), le calcul pourrait s'effectuer de manière complètement passive (« lazy » en anglais). Car la synchronisation introduite pendant l'appel de procédure est arbitraire.

3.1.2. Le gestionnaire de tâches

Modèle monolithique

On décrit un premier gestionnaire de tâches monolithique (s'exécutant sur un seul processeur) dans la figure V-1.

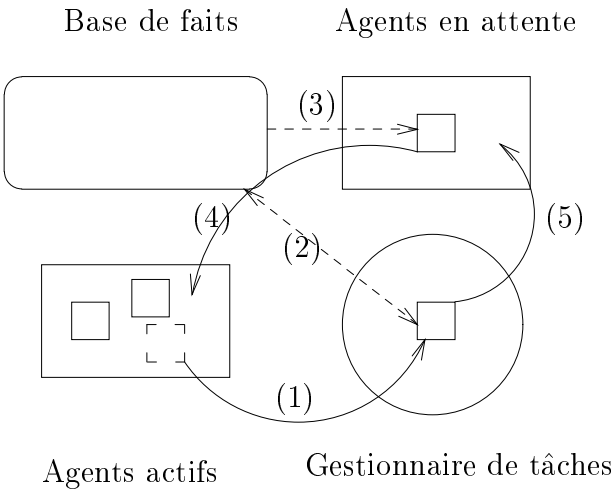


FIG. V-1: Gestionnaire de tâches monolithique

Le gestionnaire de tâches gère donc des agents qui peuvent être actifs ou suspendus. l'exécution du programme est un entrelacement d'actions atomiques. Lorsqu'un agent est activé (flèche 1), celui-ci peut interagir avec la base de faits (flèche 2). Cette interaction peut activer un agent suspendu (flèche 3), qui redevient alors actif (flèche 4). Elle peut aussi se traduire par la suspension de l'agent, qui devient donc suspendu (flèche 5).

Modèle réparti

Le modèle précédent peut être réparti sur plusieurs processeurs, à condition de centraliser les accès aux agents actifs et aux agents en attente. Cet éclatement est explicité dans la figure V-2.

L'intérêt d'un tel modèle est qu'il n'y a pas de moniteur[2] sur la base de faits. Il y a donc deux moniteurs; le premier gère les agents actifs et leurs entrelacements. Le deuxième gère la suspension des agents. Il est activé quand un agent doit être suspendu et quand une écriture peut réveiller un agent, c'est-à-dire quand on écrit pour la première fois sur une variable.

[2]Un moniteur est un point de contrôle qui n'accepte qu'un seul utilisateur à la fois. Il est réalisé avec des sémaphores.

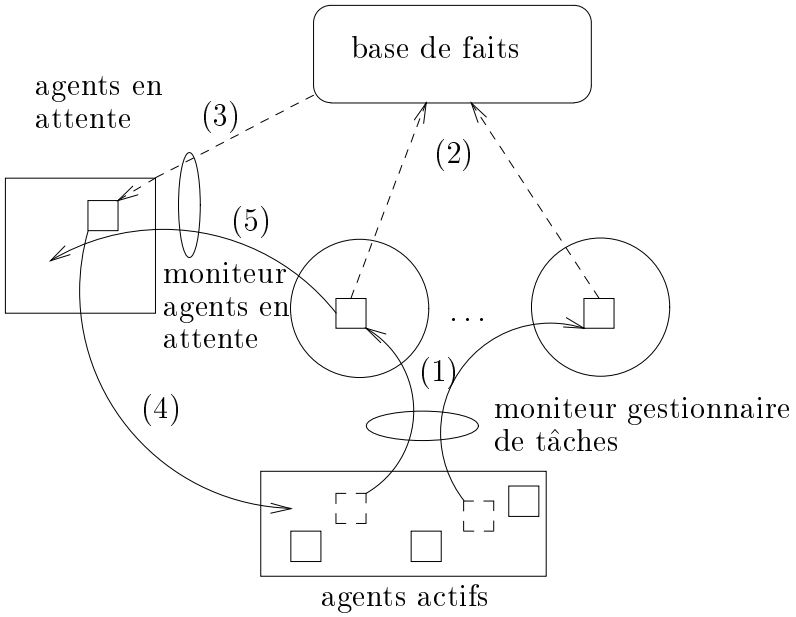


FIG. V-2: Gestionnaire de tâches réparti

3.2. Un modèle de mémoire à deux niveaux

Le modèle du non-déterminisme introduit dans la section 4. du chapitre III introduit la notion de base de faits locale. Cette notion qui accompagne la recherche encapsulée [95, 96] est raffinée pour le langage CC($\mathcal{M}$) afin d'augmenter son pouvoir d'expression.

3.2.1. PRÉSENTATION DU MODÈLE

On s'intéresse dans cette section à la gestion de la mémoire du non-déterminisme. Plutôt que de copier la méthode traditionnelle qui cherche à transporter le schéma séquentiel de la WAM dans un cadre parallèle au prix d'une gymnastique complexe et coûteuse[3], le langage CC($\mathcal{M}$) s'inspire du standard POSIX[87] pour la gestion des processus légers (« threads » en anglais). De même que le standard POSIX propose des données locales à chaque agent, le langage CC($\mathcal{M}$) propose une mémoire à deux niveaux, un niveau global, partagé par tous les agents et un niveau local qui est soit dupliqué entre l'agent fils et son père, soit dupliqué lors de la création de l'agent fils. Cette technique a deux avantages, d'une part elle simplifie considérablement la gestion des environnements lors d'un changement de contexte entre deux agents à l'exécution, d'autre part elle s'implante de manière efficace et originale.

Chaque agent se voit donc attribuer un segment de mémoire qui est mis en place lors de l'installation de l'agent en vue de son exécution comme le montre la figure V-3.

[3]La méthode traditionnelle, qui cherche à reconstruire pour chaque agent une pile d'exécution cohérente, doit s'adapter à une pile linéaire où sont stockées les variables du calcul et d'autres étrangères au calcul. Dans ce cas-là, il devient rentable d'introduire une « granularité » dans cette pile et de ne copier que les parties utiles lors de l'installation d'agents en vue de leur exécution. Cette opération

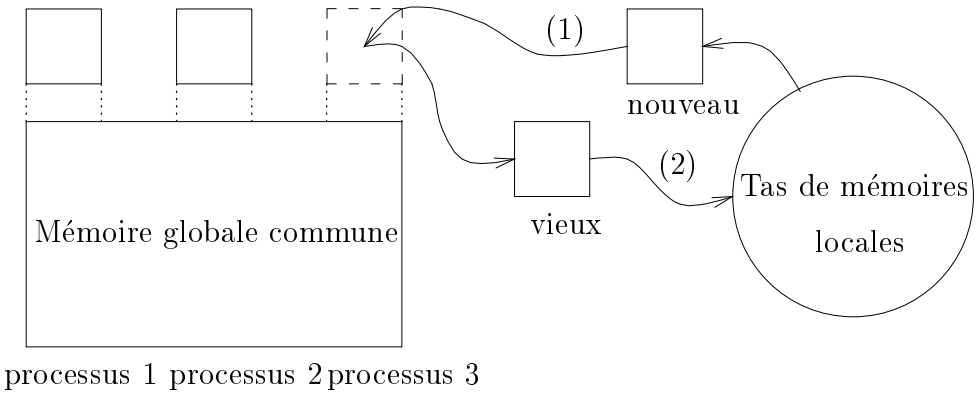


FIG. V-3: Modèle de mémoire de CC($\mathcal{M}$)

3.2.2. UTILISATION DU MODÈLE DE MÉMOIRE À DEUX NIVEAUX

L'utilisation de cette mémoire à deux niveaux se découpe en deux parties distinctes, la première au niveau de l'allocation des objets, la deuxième au niveau des agents.

Gestion de la mémoire locale des agents

La mémoire locale contient les variables d'un calcul non-déterministe. On en déduit donc que la création d'un agent peut se faire selon deux manières. Soit l'agent partage le segment mémoire de son père. Dans ce cas-là, les deux agents coopèrent ensemble sur le même segment mémoire. Soit l'agent fils duplique le segment mémoire de son père et utilise son propre segment mémoire. Dans ce cas-là, les deux calculs, ceux du père et ceux du fils sont indépendants. On obtient bien l'indépendance des deux agents sur les variables locales.

Allocation locale d'objets

La mémoire locale est dupliquée et individualisée lors de la création d'agents lors d'un point de choix non-déterministe. Celle-ci doit contenir toutes les variables libres du calcul en cours. Par contre, le but étant de minimiser les allocations mémoire sur le tas local, on gardera dans le segment mémoire global toutes les définitions statiques, ainsi que des variables résultats sur lesquelles les agents issus des différents points de choix empileront leurs résultats. Ce qui permettra de collecter à la manière d'un **bagof** les résultats d'une recherche non-déterministe.

REMARQUE V-7.
L'introduction d'un modèle mémoire à deux niveaux n'a pas la sémantique claire du modèle de mémoire suggéré par la décoration. En effet, du fait de l'existence de variables communes, le programme devient sensible aux effets de bords et à l'entrelacement des agents. Ce qui est finalement normal si on considère la procédure d'optimisation par partage et évaluation où l'on a besoin d'une variable globale qui recueille la borne courante.

appelée « copie sélective » est présentée dans [2].

4. Aperçu de l'implantation de CC($\mathcal{M}$)

4.1. Un langage écrit et compilé en C

Le langage CC($\mathcal{M}$) est implanté sur les stations Sun sous Solaris ou SunOS et sur les PC tournant sous Linux. L'implantation du langage se présente sous la forme d'un noyau en C auquel se rajoute une couche écrite en CC($\mathcal{M}$) qui est traduite vers le C par le compilateur écrit en CC($\mathcal{M}$). La taille de ces différents modules est de 6500 lignes de C pour le noyau, 7400 lignes de CC($\mathcal{M}$) pour la couche supérieure et 1400 lignes de description de la grammaire.

Parmi les modules de CC($\mathcal{M}$), on trouve le vérificateur de type, l'interpréteur, le gestionnaire de modules et le compilateur.

4.2. Implantation des objets et des messages

L'implantation des objets réutilise les travaux existants sur les langages orientés-objets, en particulier Claire[21], Objective-Caml[69].

4.2.1. Implantation des objets

Un objet est, comme dans Objective-Caml, une structure qui connaît sa classe. Donc pour implanter un objet, il suffit de réaliser une structure à laquelle on rajoute deux champs, le premier `is_a` est un pointeur de type *Classe* et le deuxième `oid` est une valeur entière qui indique une entrée dans une table de noms et donc stocke le nom de l'objet.

Par exemple, soit la structure suivante :

```
Class      <: Type [
                    is_a       : Class,
                    oid        : Integer,
                    id         : Class,
                    superclass : Class,
                    length     : Integer,
                    attributes : BinaryTree of FeatureInfo
                   ]
```

Celle-ci se retrouve compilée en la structure C suivante :

```
struct class_Class {
  struct class_Class * is_a;
  int oid;
  struct class_Class * id;
  struct class_Class * superclass;
  int length;
  struct class_BinaryTree * attributes;
};
```

4.2.2. Implantation du typage

Le typage est réalisé par un automate qui parcourt récursivement l'arbre syntaxique d'un programme et qui calcule une information de type qui est à la fois synthétisée et héritée.

L'information circule dans les deux sens. Les déclarations de type (des variables et des paramètres) descendent dans l'arbre syntaxique et le type effectif des sous-expressions remonte ce même arbre pour typer les expressions.

Le typage est réalisé par le module de vérification de type, implanté en CC($\mathcal{M}$) qui est en fait un appel récursif à des fonctions de typage surchargées `TypeIns` ou `TypeArg` selon que l'on type une instruction ou un argument (une instruction n'a pas de valeur retour).

On peut donner un exemple simple :

```
fun TypeIns(I:InsWhile, ck:Checker, mod:ModuleInfo):ActionResult :=
   var next:List
   TypeArg(test(I), ck, mod)
   next <- code(I)
   while Not ( next = Nil ) do
      if ( TypeIns(head(next), ck, mod) = WRONG ) then
         return WRONG
      else
         next <- tail(next)
      endif
   end
   return OK
end
```

La fonction précédente vérifie le type d'une instruction `While <test> Do <codes> End`. Elle vérifie d'abord le type du test puis ceux de la liste d'instructions qui lui est attachée.

`ck` de type `Checker` représente les informations de type synthétisées. La classe `Checker` contient les champs suivant :

```
Checker <: Information [args : List of Type,
                        vars : List of Type,
                        cases : List of CaseCell,
                        globalvars:List of GlobalVarDecl ]
```

où le champ `args` renseigne sur le type des arguments de la fonction que l'on est en train de typer, et le champ `vars` renseigne sur le type de variables locales. Le champ `globalvars` s'occupe des variables globales. Enfin, le champ `cases` incarne les informations qui sont calculées lors du typage, par exemple si on teste si un objet est égal à un autre de type plus petit, alors on peut déduire un meilleur type pour cet objet dans la branche *si oui* de l'instruction `If`.

`mod` est de type `ModuleInfo` et gère les modules. Il s'occupe de la portée des identificateurs.

4.2.3. Gestion des messages

La gestion des messages s'occupe des lectures et des écritures sur la mémoire. Il y a deux choses à vérifier. Si on lit une valeur, on doit suspendre l'agent si la valeur est indéfinie. Si on écrit une valeur, on doit réveiller les agents qui peuvent être suspendus en attente de cette même valeur.

Ces deux actions sont implantées en C par les deux fonctions suivantes :

```
EXTERN ref ReadAttrS(struct class_Top *obj, struct class_Feature *fea) {
  struct class_FeatureInfo *feai;
  int offset;
  ref res;
  if (obj==NULL)
    return NULL;
  feai=GetFeature(obj->is_a, fea); /* recupere la descrition du champs */
  if (feai == NULL) {              /* attachee a la classe de l'objet. */
    struct class_UnknownFeature * err;
    CREATE(err, UnknownFeature);
    err->oid = -1;
    err->on  = obj->is_a;
    err->id  = fea;
    RaiseException (err);     /* leve une exception en cas d'attribut */
  }                           /* pas defini */
  offset = feai->offset;
  CheckUndefined(obj+offset); /* verifie si la valeur est definie, /*
  return ATA(obj, offset);    /* suspend sinon */
}

EXTERN void WriteAttrS(struct class_Top * obj,
               struct class_Feature * fea,
               ref val) {
  struct class_List * P=Nil;
  struct class_FeatureInfo *feai;
  int off;
  int offset;
  struct class_List * next;
  if (obj==Nil)
    return ;
  feai=GetFeature(obj->is_a, fea);
  if (feai == NULL) {
    struct class_UnknownFeature * err;
    CREATE(err, UnknownFeature);
    err->oid = -1;
    err->on  = obj->is_a;
    err->id  = fea;
    RaiseException (err); /* leve une exception */
  }
  off=feai->offset;
  oldval = ATA(obj, off);       /* oldval = ancienne valeur */
  ATA(obj, off)=val;            /* ATA(obj, off)=offset off de l'objet obj */
  if (oldval == UNDEFINED)      /* si des agents peuvent etre reveilles */
    RestartUndefined(obj+off);  /* alors on les reveille. */
}
```

4.3. Implantation du modèle d'exécution

Le modèle d'exécution de CC($\mathcal{M}$) s'implante en deux modules distincts, le gestionnaire de tâches et le gestionnaire de mémoire

4.3.1. Implantation du gestionnaire de tâches

Le gestionnaire de tâches global gère une file d'attente des agents actifs et un tas d'agents suspendus. Il exporte plusieurs fonctions. Celles-ci sont de plusieurs natures : gestion de agents actifs (`GetFreeThread` et `PutFreeThread`), gestion des agents bloqués par les fonctions (`InsertBlockedProcess` et `PromoteBlockedProcess`) et la création de nouveaux agents (`AddProcess`). Enfin chaque gestionnaire de tâches local gère les changements de contextes entre agents et utilise l'interface offerte par le gestionnaire de tâches global.

Le gestionnaire de tâches est implanté par la classe `Scheduler` qui est définie comme suit :

```
Scheduler   <: Information [
                             hidden processes:Process,
                             index:Integer,
                             hidden waitingprocesses:Array,
                             hidden semaphore0:Integer,
                             hidden semaphore1:Integer,
                             bound:Integer,
                             hidden blocked:BlockedProcess,
                             active:Integer,
                             reserved:Integer
                           ]
```

Dans cette classe, les agents actifs forment une liste chaînée pointée par le champ `processes`. `index` est un compteur d'identificateur d'agents. Les agents suspendus sont stockés dans une table de Hachage pointé par `waitingprocesses`. Les champs `semaphore0` et `semaphore1` participent à l'implantation du moniteur sur le gestionnaire de tâches et enfin les champs `bound`, `blocked` et `reserved` implantent un système de réservation de ressources qui sera utilisé dans l'implantation du problème du pont dans le chapitre suivant.

Enfin chaque gestionnaire de tâches local implante l'algorithme suivant :

```
New <- NextThread()         (* Demande le prochain agent en attente  *)
if (New != Nil) then        (* New = Nil -> pas d'agents en attente  *)
   Old <- CurrentThread()   (* Old <- agent remplace                 *)
   EnqueueThread(Old)       (* Il est remis sur la liste d'attente   *)
   InstallLocalMemory(New)  (* La memoire de New est installee       *)
   Switch(Old, New)         (* On change de contexte                 *)
   Continue(New)            (* Puis on continue l'execution de  New  *)
endif
```

4.3.2. Implantation de la gestion mémoire

On réexpose les idées de [80].

MMU

Cette implantation utilise un microprocesseur annexe de l'unité arithmétique et logique, appelé « Memory Management Unit » dont le rôle est de traduire des adresses mémoires virtuelles en adresses physiques comme le montre la figure V-4.

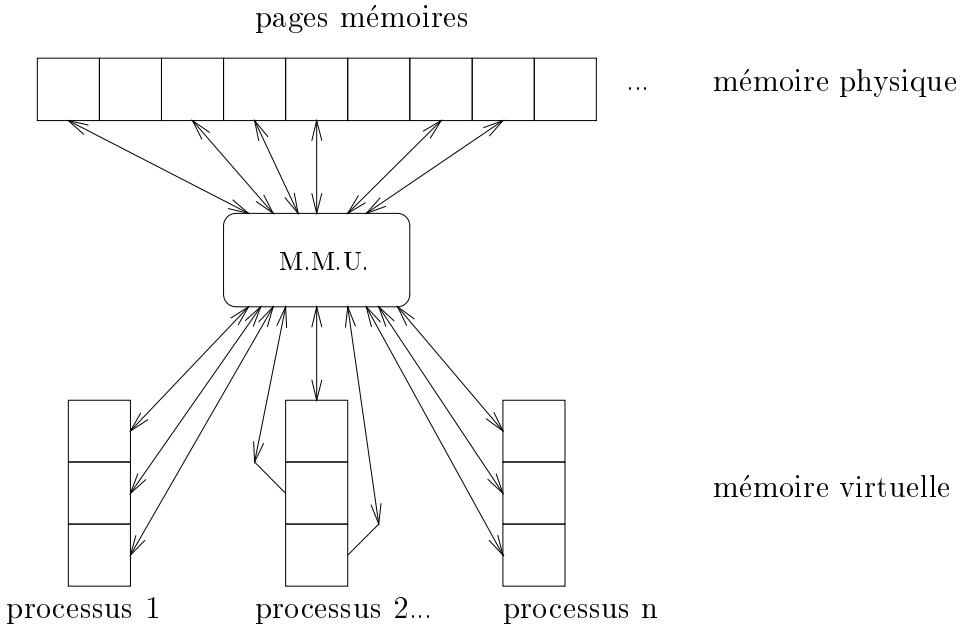


FIG. V-4: Description de la M.M.U.

Librairie I.P.C.

Suite aux progrès des systèmes d'exploitation, on peut trouver une librairie des systèmes d'exploitation Unix appelée I.P.C. (Pour « Inter-Process Communications ») qui permet d'exploiter et de programmer cette puce. Cette librairie offre trois primitives :

```
shmget ( key, size, shmflag )
shmat  ( shmid, shmaddr, shmflag )
shmdt  ( shmaddr )
```

qui permettent de créer (`shmget`) un segment de mémoire de taille `size` avec une clef `key` qui peut être publique. Cette fonction retourne un identificateur `shmid` qui est utilisé par la suite pour installer le segment de mémoire ainsi créé à une adresse précise `shmaddr` par la fonction `shmat`. Enfin, on peut dés-installer un segment de mémoire ainsi alloué grâce à la commande `shmdt`.

Implantation de la mémoire locale

Grâce aux primitives précédentes, on peut directement implanter la mémoire locale. Il suffit pour cela d'allouer pour chaque agent un segment de mémoire qui est installé à une adresse fixe quand l'agent s'exécute. Ce segment est ensuite dés-installé quand l'agent s'arrête de s'exécuter.

L'algorithme implanté par les gestionnaires de tâches locaux devient donc :

```
New <- GetNewThread()      (* Demande le prochain agent en attente  *)
if (New != Nil) then       (* New = Nil -> pas d'agent en attente   *)
   Old <- CurrentThread()  (* Old <- agent remplace                 *)
   UnmapLocalMemory(Old)   (* On desinstalle sa memoire locale      *)
   PutBackThread(Old)      (* il est remis sur la liste d'attente   *)
   MapLocalMemory(New)     (* On installe la memoire locale de New  *)
   Switch(Old, New)        (* On change de contexte                 *)
   Continue(New)           (* Puis on continue l'execution de  New  *)
endif
```

4.4. Évaluation de l'implantation de CC($\mathcal{M}$)

4.4.1. Tests typiques de vitesse

On a effectué différents tests de vitesse afin d'évaluer le coût de différentes opérations simples utilisées dans un langage de programmation par contraintes. Ces mesures nous permettront de comparer le projet CC($\mathcal{M}$) avec d'autres systèmes sur la gestion du parallélisme qui est le critère d'efficacité le plus important. Ces tests sont bien sûr théoriques et ne peuvent rendre compte parfaitement des performances d'un langage comme le feraient des tests réels sur des applications complètes. Néammoins, ils apportent une information suffisante qui permet de traduire a priori les choix d'implantation du langage CC($\mathcal{M}$) et des autres modèles d'exécution des langages logiques parallèles ou des langages parallèles par contraintes.

Ces tests ont été réalisés sur une Sun SparcStation 5 avec un code compilé avec `gcc` sans optimisations.

Les résultats sont les suivants :

Opération	**coût**
Lecture, écriture de et vers la mémoire, sans indirection	0.08 μs
Lecture, écriture, 1 indirection	0.24 μs
Lecture, écriture, 2 indirections	0.36 μs
Enlever un segment mémoire, en installer un autre	178 μs
Lire une valeur dans une table de hachage	20 μs
écrire une valeur dans une table de hachage sans test d'unicité	8 μs
copier un bloc mémoire de 4096 octets	204 μs

4.4.2. Comparaison avec les modèles dits à vecteur ou avec copie

L'implantation du parallélisme-ou [99, 98, 15] nécessite trois opérations distinctes qui sont la création d'un agent, le changement de contexte et l'accès aux variables du calcul.

On peut trouver dans la littérature un premier modèle pour l'implantation des contextes multiples qui sont nécessaires au parallélisme-ou. Ce premier modèle que nous nommerons modèle avec vecteurs maintient pour chaque agent un espace mémoire à jour où sont stockées les variables. Les variables sont référencées par des liaisons superficielles (adresse exacte) ou profondes (avec une indirection en plus). Dans ces modèles, le temps d'accès aux variables est aussi rapide que dans CC($\mathcal{M}$). Le critère important est le temps de changement de contexte.

Modèles avec copie

Les liaisons superficielles[4, 84, 86] impliquent donc une copie de la mémoire lors de la mise en place d'un agent. Pour comparer ce modèle au langage CC($\mathcal{M}$), il suffit de comparer les temps de gestion de page mémoire et de copie mémoire. Si les temps sont équivalents pour 4 Ko de mémoire, le coût est constant pour le modèle de CC($\mathcal{M}$) et linéaire en la taille du segment mémoire pour les autres modèles. Ce qui implique un gain affine sur la taille du problème entre CC($\mathcal{M}$) et les langages avec copie de mémoire.

Modèles dits avec vecteurs

Les liaisons profondes[114, 115] impliquent la gestion d'un tableau de références. La notion importante pour ces implantations est la notion de distance entre deux agents, distance sur l'arbre de recherche. En effet, dans ce modèle, l'accès aux variables se fait en temps constant, puisqu'il suffit de lire l'adresse dans la table de références. Mais le changement de contexte dépend de la distance.

Par rapport au langage CC($\mathcal{M}$), l'accès aux variables est un tout petit peu plus rapide dans CC($\mathcal{M}$) puisqu'il y a une indirection en moins. Par contre, pour des petits problèmes, la gestion du tableau est légèrement plus rapide pour les modèles avec vecteur. En pratique, étant donné que le sur-coût est constant pour CC($\mathcal{M}$) et linéaire pour les autres langages, on préférera le langage CC($\mathcal{M}$) qui assurera des performances indépendantes de la taille du problème.

4.4.3. Comparaison avec les modèles avec tables de hachage

Un deuxième modèle répandu de contextes multiples utilise des tables de hachage attachées à chaque point de choix pour stocker la différence entre la base de faits locale et la base de faits globale. Ces modèles ont été utilisés dans AKL [64], dans Oz [102, 104] ou encore dans des langages logiques parallèles [27, 28], Argonne [11], PEPSys [5]. En fait, la mémoire se décompose en un ensemble de tables de hachage disséminé sur l'arbre de recherche avec à sa racine la mémoire initiale.

Si l'on compare maintenant le langage CC($\mathcal{M}$)à ces modèles, il est facile de voir que le sur-coût induit par l'accès aux variables dépasse le temps de changement de contexte. En fait, il suffit de 9 (178 / 20) opérations de lecture (en supposant qu'elles n'utilisent que la table de hachage du premier point de choix, sans remonter à des points de choix précédents) pour contrebalancer le coût de la gestion des segments mémoires. Et il est clair que 9 lectures, pendant la fenêtre de temps allouée à un agent dans une implantation à temps partagée, est un critère raisonnable. C'est pourquoi l'implantation de CC($\mathcal{M}$) est plus rapide que celles issues du modèle avec tables de hachage.

Chapitre VI

Implantation et évaluation de CC(X)

Le but de cette partie est de démontrer les capacités du langage cc($\mathcal{M}$)à implanter des langages de contraintes concurrents. Il s'agit donc de montrer que de par sa conception adaptée, il est aisé de coder des domaines de contraintes divers au dessus de cc($\mathcal{M}$). Si les constructeurs du langage simplifient la tâche d'implantation d'un domaine de contraintes en offrant une puissance d'expression suffisante tout en gardant extensibilité et efficacité, alors le but du projet cc($\mathcal{M}$) est atteint.

1. Le langage cc ($\mathfrak{B}$)

1.1. Présentation du domaine de contraintes des booléens

1.1.1. Les contraintes atomiques

Nous nous intéressons ici à un domaine très simple puisqu'il s'agit de celui des booléens. Dans ce domaine, on a deux valeurs pré-définies : **vrai** et **faux**. Les contraintes atomiques sont définies sur un ensemble $\mathcal{V}$ de variables et sont de quatre types. Soient X, Y, Z trois éléments de $\mathcal{V} \cup \{\mathbf{vrai}, \mathbf{faux}\}$, les contraintes booléennes atomiques[1] sont :

- $\mathbf{and}(X, Y, Z)$
- $\mathbf{or}(X, Y, Z)$
- $\mathbf{not}(X, Y)$
- $\mathbf{eq}(X, Y)$

On dénote par $\mathfrak{B}$ l'ensemble de toutes les contraintes booléennes.

1.1.2. Règles de propagation

On s'intéresse maintenant aux règles de propagation entre contraintes booléennes. Celles-ci sont déclenchées lorsqu'une valeur est attribuée à une variable. Par souci de concision, nous ne traiterons que le cas de la contrainte $\mathbf{and}(X, Y, Z)$ dont la signification est :

$$Z = \mathbf{and}(X, Y)$$

[1] Il existe bien sûr d'autres représentations des mêmes contraintes, mais on peut décider d'utiliser celle-là. Il est clair qu'elles sont toutes équivalentes. On en choisit une qui est habituelle.

Pour les autres contraintes, nous invitons le lecteur à se rapporter à l'article de référence [32] dont ce travail est très largement inspiré.

- $X = \textsf{faux} \rightarrow Z = \textsf{faux}$.
- $Y = \textsf{faux} \rightarrow Z = \textsf{faux}$.
- $X = \textsf{vrai} \rightarrow Z = Y$.
- $Y = \textsf{vrai} \rightarrow Z = X$.
- $Z = \textsf{vrai} \rightarrow X = \textsf{vrai}$.
- $Z = \textsf{vrai} \rightarrow Y = \textsf{vrai}$.

Exemple VI-1.: and(X, Y, Z)

1.1.3. Une représentation simple

Dans une optique *boîte transparente* (en anglais « glass box »), nous allons coder toutes ces règles de propagation à l'aide d'une règle unique de bas niveau [32] :

$$l_0 <= l_1, \ldots, l_n$$

où l_i est soit X (un littéral positif), soit $-X$ (un littéral négatif). La sémantique de cette règle est la suivante : dès que tous les $l_1 \wedge \ldots \wedge l_n$ sont vrais alors l_0 est vrai. De plus, dans le domaine qui nous intéresse avec les contraintes atomiques que nous avons choisies, nous pouvons nous restreindre à $n = 1$ ou $n = 2$.

Nous allons ainsi pouvoir proposer des règles simplifiées de propagation grâce à cette règle unique. Nous allons encore une fois ne traiter que le cas de la contrainte atomique **and**. Pour les autres contraintes atomiques, nous prions le lecteur de se rapporter encore à [32].

---

and(X, Y, Z) $\vdash$

$$\begin{array}{rcl} Z & <= & [X, Y] \\ -Z & <= & [-X] \\ -Z & <= & [-Y] \\ X & <= & [Z] \\ -X & <= & [Y, -Z] \\ Y & <= & [Z] \\ -Y & <= & [X, -Z] \end{array}$$

Exemple VI-2.: Règles simplifiées pour **and**

1.2. Implantation en CC($\mathcal{M}$)

Nous passons maintenant à l'implantation proprement dite du domaine de contraintes des booléens en CC($\mathcal{M}$).

1.2.1. Implantation des variables et dépendances

Il convient dans un premier temps d'implanter les contraintes et les dépendances entre les variables booléennes. Pour cela, nous utiliserons deux structures particulières, la première (`Booleanvariable`) servira à coder les variables. Cette structure contiendra trois champs importants, le premier étant la valeur de la variable (si elle est définie), les deux autres pointeront vers deux listes de la deuxième structure. Ces deux listes seront en fait les dépendances positives et négatives attachées à cette variable.

Les dépendances sont donc encodées avec la deuxième structure (`RecordFrame`). Celle-ci code la règle

$$l_0 <= l_1, l_2$$

Le cas $l_0 <= l_1$ sera un cas particulier où la deuxième variable (le champs `othervariable`) sera égale à `nil`. Cette structure possède donc quatre champs, la variable à modifier si la règle de propagation est activée, ainsi que la valeur à affecter. Les deux autres champs sont liés à la deuxième variable (terme l_2 de la règle de propagation). Il montre la variable à utiliser ainsi que la valeur attendue.

```
BooleanDomain <: Top []

BooleanValue <: BooleanDomain []
BooleanVariable <: BooleanValue [
   positivedependance : RecordFrame,
   negativedependance : RecordFrame,
   value : Bool
]
BooleanConstant <: BooleanValue [value : Bool]

RecordFrame <: BooleanDomain [
   nextrecord : recordFrame,
   targetvariable : BooleanVariable,
   targetvalue : Bool,
   othervariable : Booleanvariable,
   otherrequiredvalue : Bool
]
```

On peut faire deux remarques sur cette implantation. La première concerne les valeurs fixes (`BooleanValue`), celles-ci peuvent être toujours substituées dans une contrainte à une variable. La deuxième remarque concerne les dépendances liées à une règle contenant deux littéraux à droite du <=. Dans ce cas-là, pour plus de symétrie, il convient de générer deux structures pour chacun des deux littéraux (si ce sont des variables) et de faire pointer le champ `othervariable` vers l'autre variable. Mais dans ce cas-là, il convient d'attacher ces deux structures entre elles afin de lier leur sort. Mais ce n'est pas traité dans l'article original qui la décrit. Cela prouve que cette méthode d'implantation des booléens n'est pas optimale et qu'il reste des améliorations à apporter.

1.2.2. Implantation des contraintes booléennes

Maintenant on peut définir les contraintes booléennes de haut niveau.

```
BooleanConstraint <: BooleanDomain []

AndConstraint <: BooleanConstraint [
   left : BooleanValue,
   right1 : BooleanValue,
   right2 : BooleanValue   (* left = And(right1, right2) *)
]

EqConstraint <: BooleanConstraint [
   left : BooleanValue,
   right : BooleanValue
]
...
```

Maintenant que tout est implanté avec une règle de bas niveau unique, rien ne nous empêche de définir d'autres contraintes atomiques comme **xor**, **nand**, **nor** et ainsi de suite.

1.2.3. Implantation du *tell*

Chaque domaine de contraintes maintenant exporte des fonctions génériques qui peuvent être utilisées dans des algorithmes standards. Ces deux fonctions sont ***ask*** et ***tell***. Nous implantons d'abord le ***tell*** sur le domaine des booléens.

La première étape est de définir les fonctions qui posent les règles de bas niveau. Ces fonctions sont `tell_crst1` et `tell_crsr2`.

La fonction `tell_crst1` pose la contrainte $l_0 <= l_1$.

```
fun tell_crst1 ( left:BooleanVariable,
                 writtenleft:Bool,
                 right:BooleanValue,
                 expectedright:Bool ) : Bool := ...

                 (*** construit la bonne structure ***)
```

Le code est assez explicite. Il teste si la variable l_1 a déjà une valeur définie. Dans ce cas-là, il écrit la valeur de l_0 s'il y a propagation. Si la variable l_1 n'a pas de valeur, il crée la structure `RecordFrame` et l'initialise avec les bonnes valeurs. La liste des dépendances sera parcourue par la fonction `WriteVar`.

De manière similaire, on définit `tell_crst2` pour la règle $l_0 <= [l_1, l_2]$.

```
fun tell_crst2 ( left:BooleanVariable,
                 writtenleft:Bool,
                 right1:BooleanValue,
                 expectedright1:Bool,
                 right2:BooleanValue,
                 expectedright2:Bool ) : Bool := ...

                 (*** construit la bonne structure ***)
```

Nous faisons maintenant la conversion entre les deux niveaux de représentation. Nous utiliserons une fonction générique : `WriteVar` qui prend deux arguments, une variable et une valeur. Pour le domaine des booléens, celle-ci est définie par `WriteVar(X :BooleanVariable, V :BooleanConstant)`. Cette fonction exécute deux actions. La première est de tester si la valeur est compatible avec la variable (c'est-à-dire si la variable a déjà une valeur qui lui est affectée). Si les deux valeurs sont incompatibles, la fonction tue l'agent en cours ou lève une exception au choix. Dans l'autre cas, elle parcourt la liste de dépendance et se réappelle récursivement en cas de propagation. Grâce à cette fonction, nous pouvons définir toutes les méthodes `tell` sur les différentes contraintes du domaine :

```
fun tell(C:AndConstraint):Bool :=  (* On suppose que les trois arguments sont
                                      des variables *)

  tell_crst2(left(c), true, right1(c),
             true, right2(c), true)               (*  Z <= [X,Y] *)
  tell_crts1(left(c), false, c.right2, false)     (* -Z <= [-Y]  *)
  ...
end

fun tell(C:EqConstraint):Bool :=
  if (is_a(left(c)) = BooleanConstant) then
    if (is_a(right(c)) = BooleanConstant) then
      if (value(left(c)) = value(right(c)) then
        return true
      else
        raise fail                                (* true = false *)
      endif
    else
      WriteVar(value(right(c)), value(left(c)))   (* affectation *)
    endif
  else
    if (is_a(right(c)) = BooleanConstant) then
      WriteVar(value(left(c)), value(right(c)))   (* affectation *)
    else
      tell_crst1(left(c), true, right(c), true)   (*  X <= [ Y] *)
      tell_crst1(left(c), false, right(c), false) (* -X <= [-Y] *)
      tell_crst1(right(c), true, left(c), true)   (*  Y <= [ X] *)
      tell_crst1(right(c), false, left(c), false) (* -Y <= [-X] *)
    endif
  endif
  return true
end
...
```

1.2.4. Implantation du *ask*

On peut implanter de manière simple la règle du ***ask***. Par exemple, pour demander une contrainte **and**, il suffit d'appeler la méthode `ask` suivante :

```
fun ask(C:AndConstraint): Bool :=
  if (value(left(c)) = and(value(right1(c)), value(right2(c))) then
    return true
  else
    return false
  endif
end
```

Celle-ci repose sur les mécanismes de synchronisation de CC($\mathcal{M}$) et attend que tout soit défini pour répondre. C'est une implantation naïve mais qui marche. Elle peut bien sûr être raffinée et répondre **faux** plus tôt.

1.3. Conclusion sur CC ($\mathfrak{B}$)

L'implantation du domaine de contraintes des booléens au dessus de CC($\mathcal{M}$) est finalement facilitée par la conception du langage. Ceci se voit d'après deux critères. Le premier est la facilité avec laquelle nous avons pu représenter le type abstrait de données correspondant à la représentation bas niveau du domaine dans CC($\mathcal{M}$). Ceci est la conséquence de deux facteurs, les classes et les objets de CC($\mathcal{M}$) d'un côté et la possibilité de surcharge des méthodes issue du paradigme objet. Grâce à cela, nous avons pu implanter un domaine de contraintes particulier qui se conforme à une interface fixe[2]. De plus, il est facile, toujours en utilisant la surcharge et la sélection des méthodes d'après le type des objets, d'écrire des algorithmes génériques qui utiliseront ce domaine de contraintes en particulier ou tout autre domaine se conformant à la même interface en général.

Le deuxième critère est un critère de découpage du travail. À aucun moment dans l'implantation du domaine de contraintes des booléens, nous ne nous sommes intéressés à la gestion des agents, du parallélisme et à la synchronisation entre les agents. Ceci est particulièrement visible dans le codage de la méthode `ask`.

Ainsi donc nous avons montré que le projet CC($\mathcal{M}$) répond au premier besoin d'un implanteur de langage de contraintes concurrent, ce besoin étant la facilité de mise en œuvre. Il s'agit de montrer maintenant le deuxième besoin qui est un besoin d'efficacité. Pour cela nous avons implanté quelques tests de vitesses sur les booléens que l'on peut trouver dans [32] : Les reines 8x8 et le lemme de Shur pour n = 10, 13, 30. Les résultats sont les suivants :

Nombre de processeurs	1	2
Reines 8	490 ms	248 ms
Gain :	1	1.97
Retours en arrière :	735	735
Schur 10	490 ms	250 ms
Gain :	1	1.97
Retours en arrière :	731	731
Schur 13	540 ms	270 ms
Gain :	1	1.98
Retours en arrière :	833	833
Schur 30	540 ms	270 ms
Gain :	1	1.98
Retours en arrière :	833	833

[2]cette interface comporte trois fonctions, `tell`, `ask` et `WriteVar`

Il y a peu de commentaires à faire sur ces exemples. Les performances sont bonnes mais loin de celles obtenues par le système `clp(`$\mathcal{B}$`)`. Ceci est dû au fait que les problèmes ainsi attaqués sont très simples calculatoirement. Cette simplicité pénalise le langage `CC(`$\mathcal{M}$`)` puisque le temps de création d'un agent devient nettement supérieur à celui du calcul lors d'une étape de résolution. Il est évident que sur ces problèmes, un système séquentiel à base de retour en arrière (« backtrack ») est mieux adapté que le langage `CC(`$\mathcal{M}$`)` qui vise des problèmes avec peu de points de choix mais des calculs lourds à chaque étape de la résolution (ce qui rendrait le temps de création d'un agent négligeable).

2. CC(FD)

2.1. Introduction

Il s'agit ici de programmer dans le langage `CC(`$\mathcal{M}$`)` un fragment des domaines finis [113, 56, 112]. On cherche à résoudre des problèmes d'ordonnancements disjonctifs. Ces problèmes sont basés sur les contraintes de précédences

$$X \leqslant Y + C$$

et sur la disjonction de ces contraintes. Ce sont des problèmes de CLP classiques qui utilisent des algorithmes de partage et évaluation (« Branch and bound »). Ils peuvent bénéficier d'une implantation du non-déterminisme à l'aide d'un parallélisme-ou équitable[3].

La recherche de solutions peut se faire soit en fixant les valeurs des variables, soit en choisissant la composante réalisée d'une disjonction de contraintes de précédence. C'est cette deuxième méthode que nous allons implanter.

2.2. Implantation générique des domaines finis

2.2.1. Encodage du domaine de contraintes

Nous allons regrouper tout ce module sous la classe `ScheduleProblem`. Nous avons affaire principalement à quatre types d'objets, les variables (`ScheduleVar`), les dépendances entre ces variables (`ScheduleDepSup` et `ScheduleDepInf`), une structure de données pour représenter les disjonctions (`ScheduleChoice`) que comporte le problème et qui vont être choisies et déterminisées par l'algorithme de recherche de solutions, et la description d'un problème à résoudre avec la structure qui supporte l'algorithme de recherche de solutions (`ScheduleSolution`, `ScheduleAux`).

---

[3]En effet, soit un problème qui a deux branches possibles, la première très coûteuse qui donne une mauvaise solution et la seconde rapide qui donne une solution meilleure. Dans ce cas-là, une implantation séquentielle calcule la première solution, puis la seconde. Le temps final est donc la somme des temps des deux branches. Une implantation parallèle calcule les deux branches à la fois (deux fois moins vite), mais trouve la solution dès que la deuxième branche a fini, soit en deux fois le temps de la seconde branche, ce qui est par définition inférieur à la somme des deux temps ; le gain est donc ici sur-linéraire. La parallélisation de l'algorithme de partage et évaluation se traduit en général par une légère perte d'efficacité mais aussi par une stabilisation des performances, ce qui est appréciable. On peut trouver un article sur ce sujet dans [88, 111]

```
ScheduleProblem  <: Anonymous []

ScheduleVar <: ScheduleProblem [
                            name   : Word,
                            min    : Integer,
                            max    : Integer,
                            depsup : ScheduleDepSup,
                            depinf : ScheduleDepInf,
                            changed: Bool,
                            next   : ScheduleVar
                          ]

ScheduleChoice <: ScheduleProblem [
                             var1 : ScheduleVar,
                             var2 : ScheduleVar,
                             cst1  : Integer,
                             cst2  : Integer,
                             done  : Boolean,
                             next  : ScheduleChoice
                           ]

ScheduleAux    <: ScheduleProblem [
                                  count :Integer
                                  choices : SchedulePoint,
                                  ]

ScheduleSolution <: ScheduleProblem [
                                  bound   : Integer,
                                  aux     : ScheduleAux,
                                  vars    : ScheduleVar,
                                  ending  : ScheduleVar,
                                ]

ScheduleDepSup <: ScheduleProblem [
                          var1 : ScheduleVar,
                          var2 : ScheduleVar,
                          cste : Integer,
                          next : ScheduleDepSup
                        ]

ScheduleDepInf <: ScheduleProblem [
                          var1 : ScheduleVar,
                          var2 : ScheduleVar,
                          cste : Integer,
                          next : ScheduleDepInf
                        ]
```

Code source VI-3.: Domaine de contraintes

Comme nous pouvons le voir sur ce code source, une variable est définie par un nom, un min et un max et par deux listes de dépendances sur le min et max de cette variable. De plus, les variables sont regroupées en une liste chaînée.

Une dépendance entre variable est de la forme

$$max(X) \leqslant max(Y) + C$$

Ce qui se code en `CC(`$\mathcal{M}$`)` par les classes `ScheduleDepSup` et `ScheduleDepInf` qui comportent la variable cible `var1` (X dans l'exemple), la variable source `var2` (Y dans l'exemple) et la constante `cste` (C dans l'exemple).

Une disjonction est de la forme $(X \leqslant Y + C) \vee (Y \leqslant X + C')$. Elle traduit l'exclusion mutuelle de deux ressources avec durée d'utilisation. Elle s'implante facilement par la structure `ScheduleChoice`.

Enfin, un problème en cours de résolution possède une borne courante `bound`, une liste de choix avec un compteur annexe `aux`, une liste de variables `vars` et une variable dite finale qui donnera la durée totale du problème à ordonnancer `ending`.

2.2.2. Encodage de la fonction *tell*

Il y a deux type de contraintes, les contraintes simples et les disjonctions de contraintes. Pour poser la contrainte

$$X \leqslant Y + C$$

on appellera la fonction `AddScheduleDependance("X", "Y", C, S)` où `S` est la structure représentant le problème à résoudre.

```
fun AddScheduleDependance ( N1:Word, N2:Word, CC:Integer,
                            S:ScheduleSolution ) : ActionResult :=
   var V1:ScheduleVar
   var V2:ScheduleVar

   V1 <- FindScheduleVar ( S, N1)
   V2 <- FindScheduleVar ( S, N2)
   AddDependance ( V1, V2, CC)
   return OK
end

fun AddDependance ( X:ScheduleVar, Y:ScheduleVar, Z:Integer):ActionResult :=

   if (X = Y) then
      return WRONG
   endif

   depinf(X) <- ScheduleDepInf [
```

```
                                var1 <- X,
                                var2 <- Y,
                                cste <- Z,
                                next <- depinf(X)
                              ]

   depsup(Y) <- ScheduleDepSup [
                                var1 <- Y,
                                var2 <- X,
                                cste <- -Z,
                                next <- depsup(Y)
                              ]

   if ( max(Y) + Z < min(X) ) then
      AbortThread()
   endif
   if ( max(Y) + Z < max(X) ) then
      WriteMax ( X, max(Y) + Z )
   endif
   if ( min(Y) < min(X) - Z ) then
      WriteMin ( Y, min(X) - Z )
   endif

   return OK
end
```

Code source VI-4.: Dépôt d'une contrainte simple

---

mais on utilise aussi des contraintes disjonctives. Et donc pour poser

$$(X \leqslant Y + C) \vee (Y \leqslant X + C')$$

on appellera `AddScheduleChoice("X", "Y", C1, C2, S)`.

```
fun AddScheduleChoice ( N1:Word, N2:Word, C1:Integer, C2:Integer,
                        S:ScheduleSolution ) : ActionResult :=
   var V1:ScheduleVar
   var V2:ScheduleVar
   var CH:ScheduleChoice

   V1 <- FindScheduleVar ( S, N1)
   V2 <- FindScheduleVar ( S, N2)
   CH <- ScheduleChoice [[
                          var1 <- V1,
                          var2 <- V2,
                          cst1 <- C1,
                          cst2 <- C2,
                          done <- false,
```

```
                        next <- choices(aux(S))
                      ]]
   choices(aux(S)) <- CH
   count(aux(S)) <- count(aux(S)) + 1
   return OK
end
```

Code source VI-5.: Dépôt d'une contrainte disjonctive

2.2.3. Propagation d'une contrainte

Il y a deux fonctions qui propagent les contraintes. Elles sont appelées quand on écrit sur le max (`WriteMax`) ou sur le min (`WriteMin`) d'une variable. Ces fonctions parcourent la liste des dépendances d'une variable et s'appellent récursivement s'il y en a besoin. De plus, elles peuvent détecter une boucle (ce qui conduit à une erreur puisqu'une variable va avoir un domaine vide[4]). En cas d'erreur, l'agent se tue et laisse les autres continuer la recherche.

```
fun WriteMax ( X:ScheduleVar, Z:Integer ) : ActionResult :=
   var dep : ScheduleDepSup

   if (changed(X)) then
      AbortThread()
   endif

   changed(X) <- true
   max(X) <- Z

   dep <- depsup(X)
   while Not ( dep = Nil )  do

      if ( Z < min(var2(dep)) + cste(dep) ) then
         AbortThread()
      endif

      if ( Z < max(var2(dep)) + cste(dep) ) then
         WriteMax ( var2(dep), Z - cste(dep) )
      endif

      dep <- next(dep)
   end
   changed(X) <- false

   return OK
end

fun WriteMin ( X:ScheduleVar, Z:Integer ) : ActionResult :=
```

[4] Ce qui peut se démontrer si on considère la linéarité des contraintes.

```
   var dep : ScheduleDepInf

   if (changed(X)) then
      AbortThread()
   endif

   min(X) <- Z
   changed(X) <- true

   dep <- depinf(X)
   while Not ( dep = Nil )  do

      if ( max(var2(dep)) + cste(dep) < Z ) then
         AbortThread()
      endif

      if ( min(var2(dep)) + cste(dep) < Z ) then
         WriteMin ( var2(dep), Z - cste(dep) )
      endif

      dep <- next(dep)
   end
   changed(X) <- false

   return OK
end
```

Code source VI-6.: Propagation d'une contrainte

2.2.4. Résolution des problèmes, recherche de solution

Pour résoudre un problème, il faut choisir une disjonction, tester si l'on prend le membre gauche (`SolveSchedule`) ou le membre droit (`SolveScheduleAux`), rajouter la dépendance ainsi sélectionnée sans oublier de faire l'élagage de l'arbre en propageant la borne courante (`bound(S)`) sur les variables.

```
fun SolveSchedule ( S:ScheduleSolution ) : ActionResult :=
   var CH:ScheduleChoice

   if ( bound(solution) < max(ending(S)) - 1 ) then (* Bound *)
      WriteMax(ending(S), bound(solution) - 1)
   endif

   CH <- FindScheduleChoice ( S )
   detach local(count(aux(S))) SolveScheduleAux (S, CH)
   AddDependanceLocal (var1(CH), var2(CH), cst1(CH) )
```

```
   if ( count(aux(S)) = 0 ) then
      if (min(ending(S)) < bound(S)) then
         bound(S) <- min(ending(S))
         print ( "[%i]", min(ending(S)) )
      endif
   endif
   return OK
end

fun SolveScheduleAux ( S:ScheduleSolution,
                       CH:ScheduleChoice ):ActionResult :=

   if ( bound(solution) < max(ending(S)) - 1 ) then (* Bound *)
      WriteMax(ending(S), bound(solution) - 1)
   endif
   AddDependanceLocal (var2(CH), var1(CH), cst2(CH) )

   if Not ( count(aux(S)) = 0 ) then
      CH <- FindScheduleChoice ( S )
      detach local(count(aux(S))) SolveScheduleAux (S, CH)
      AddDependanceLocal (var1(CH), var2(CH), cst1(CH) )
   endif

   if ( count(aux(S)) = 0 ) then
      if (min(ending(S)) < bound(S)) then
         bound(S) <- min(ending(S))
         print ( "[%i]", min(ending(S)) )
      endif
   endif
   return OK
end
```

Code source VI-7.: Recherche d'une solution

Ce code source fait apparaître un argument entier devant le mot clef `local`. En effet, s'il n'y a pas de contrôle sur la création d'agents, cette implantation explose au niveau du nombre d'agents actifs. Pour gérer cela, on introduit un système de réservation lors de la création d'un agent. Si l'on sait combien d'agents un agent peut créer directement avant de terminer, alors on peut autoriser la création d'un agent si la réservation des ressources nécessaires est possible. Sinon, la création se suspend et attend la libération des ressources nécessaires. Cet algorithme a été testé avec succès sur le problème du pont. Il correspond à une limite sur la largeur d'exploration de l'arbre de choix.

2.2.5. Sélection du point de choix

La recherche de solution est paramétrée par une fonction qui choisit le point de choix qui va être exploré. On peut donner une version minimale (qui ne choisit rien) et souvent très mauvaise de cette fonction `FindScheduleChoice` :

```
fun FindScheduleChoice ( S:ScheduleSolution ) : ScheduleChoice :=
   var CH:ScheduleChoice

   CH <- choices(aux(S))
   choices(aux(S)) <- next(CH)
   count(aux(S)) <- count(aux(S)) - 1

   return CH
end
```

Code source VI-8.: Sélection du premier point de choix de la liste

Cette fonction retourne juste le premier point de choix libre du problème en cours.

2.3. Évaluation de CC(FD)

On a choisi pour montrer les performances du langage CC($\mathcal{M}$) d'implanter un problème classique pour les domaines finis, à savoir le domaine du pont. Ce problème vise à optimiser en temps la construction d'un pont. Ce chantier (qui est présenté dans [56]) comporte plus d'une cinquantaine de tâches avec chacune une durée et des contraintes de précédence entre elles.

Il existe des techniques d'implantations très rusées qui exploitent très bien les propriétés mathématiques du problème afin d'en trouver un solution très rapidement et de manière presque déterministe (de l'ordre de quelques retours en arrière). Les implanter n'aurait pas permis de tester la vitesse du langage CC($\mathcal{M}$) mais simplement son expressivité. C'est pour cela que nous avons choisi une heuristique inefficace puisque la recherche d'un solution optimale (104 jours) ainsi que sa preuve d'optimalité utilise plus de 7600 retours en arrière.

Nous avons en plus testé le gain en vitesse selon le nombre de processeurs présents. Le test s'est fait sur un PC (Cyrix P200+) sous Linux en utilisant gcc avec l'option -O2. Les tests de multiprocesseurs sont simulés en utilisant plusieurs processus différents sur le même processeurs. Cette simulation a déjà été utilisée et comparée à du vrai parallélisme et elle a été trouvée correcte et donnant des résultats fiables.

nombre de processeurs	1	2	3	4
Preuve d'optimalité	3.32 s	1.67 s	1.15 s	0.86 s
Gain	1	1,98	2,88	3,86
Solution optimale	0.27	0.27	0.27	0.27
Gain	1	1	1	1
Nombre de retours en arrière	7600	7600	7600	7600

Ces tests sont à comparer aux implantations traditionnelles par exemple en Claire.

	Claire naif	**Claire optimisé**
Solution optimale	106 ms	30 ms
Première solution	10 ms	30 ms
Preuve d'optimalité	660 ms	10 ms
Retours n arrière	4008	0

Ces tests de vitesse appellent deux remarques. La première est que la vitesse du langage CC($\mathcal{M}$) est bonne : 7600 retours en arrière en 3.32 secondes, compte-tenu que chaque hypothèse testée implique la création d'un agent, indique une implantation performante. Bien sûr, on a un rapport constant (.2.5) avec une implantation séquentielle du fait même de la technique d'implantation à base de processus légers, mais cette implantation est capable d'exploiter des machines multi-processeurs, ce qui compense le sur-coût.

La deuxième remarque à faire juge le gain en fonction du nombre de processeurs. Même si celui-ci est bon, il traduit un léger ralentissement lié à la cohabitation de plusieurs processus de calcul en même temps. Ce ralentissement est dû aux processus bloqués sur des sémaphores communs, ceux-ci sont de deux types, sémaphores sur la mémoire[5] et sémaphore sur le gestionnaire de tâche. C'est ce dernier qui est responsable du caractère sous-linéaire du gain. A l'heure actuelle, il est difficile de dire si ce gain peut être amélioré par une optimisation de l'implantation d'un gestionnaire de tâches partagé par tous les processeurs. Il y a quand même un espoir possible. Il vient des implantations des processus légers dans le noyau même du système d'exploitation (LWP sous Solaris, Threads en mode noyau sous Linux). Avec de telles techniques, on peut imaginer une amélioration du caractère linéaire du gain de l'implantation du langage CC($\mathcal{M}$) sur des machines multi-processeurs.

Enfin, il faut remarquer que sur le probleme du pont, on est capable de trouver la solution optimale et de la prouver sans retours en arrière. Dans une telle implantation, les temps du langage CC($\mathcal{M}$) sont évidemment bon mais cela ne montre pas l'efficacité de la gestion de non-déterminisme telle qu'elle est implantée dans le langage CC($\mathcal{M}$).

2.4. Conclusion sur CC(FD)

Là encore, le langage CC($\mathcal{M}$) montre son pouvoir d'expression et la facilité avec laquelle on peut implanter des systèmes de contraintes généraux dans ce langage. Mais la conclusion de cette section se situe au niveau de l'efficacité. En effet, comme on a pu le voir, l'implantation de langage CC($\mathcal{M}$) est efficace. Le code généré par le compilateur est très proche d'un code C écrit par un programmeur humain. La vitesse d'exécution d'une procédure est donc comparable avec celle écrite dans un langage impératif classique.

De plus, le surcoût dû à une implantation parallèle ou pseudo-parallèle est absorbé par le gain de puissance sur des machines multi-processeurs. Ce qui justifie pleinement cette implantation.

En résumé, l'efficacité de l'implantation du langage CC($\mathcal{M}$) justifie deux choix qui ont été faits pendant sa conception. Le premier est lié à une architecture en couche et à un langage noyau. Ce choix s'est révélé payant puisque le langage CC($\mathcal{M}$) est un langage de plus haut niveau que le C avec des performances comparables. On a donc simplifié la tâche du programmeur en lui fournissant un langage adapté (existence du non-déterminisme, des objets, de la surcharge des fonctions et de la composition parallèle).

Le deuxième choix qui est ainsi justifié est lié au parallélisme. Puisque le surcoût est inférieur au gain dès que l'on a deux processeurs sur la station de travail, le choix d'une implantation parallèle est lui aussi justifié. En effet, le langage CC($\mathcal{M}$) offre les avantages du parallélisme (gain de puissance, réactivité, recherche équitable de solution) pour un coût limité.

[5] Celui-ci n'intervient pas dans ce test puisque tous les blocs mémoires alloués par le programme pendant l'exécution sont de type local et donc n'intéressent que l'agent actif.

Chapitre VII

Conclusion et perspectives

1. Comparaison avec des travaux proches

1.1. Remarques sur le projet CC($\mathcal{M}$)

Le modèle d'exécution de langage CC($\mathcal{M}$) est un modèle simple qui peut se comparer avec les autres modèles existants (modèle des langages logiques parallèles, modèle d'Andorra ou modèle d'Oz). Avant de faire des comparaisons spécifiques, il convient de faire les remarques suivantes.

1.1.1. Un langage non basé sur la WAM

Il faut noter que le langage CC($\mathcal{M}$) n'est pas basé sur la WAM. S'il s'inspire des travaux sur les langages logiques parallèles pour son implantation, il s'affranchit complètement des contraintes induites par la parallélisation de la WAM en proposant un modèle de mémoire à deux niveaux. Ce changement traduit une philosophie très différente de la programmation, dans le sens où la gestion du non-déterminisme repose sur le programmeur au travers des variables locales et globales. Le non-déterminisme n'est plus un ajout transparent à un langage existant. C'est au contraire un élément important dont il faut tenir compte dans la conception des algorithmes et des structures de données.

1.1.2. Un langage sans gardes

Il faut aussi noter que l'opérateur $\oplus$ n'introduit pas de garde dans le langage. En particulier, le langage CC($\mathcal{M}$) n'a pas de garde profonde. Cette différence avec les langages logiques parallèles, Oz et AKL, est justifiée par le but du projet CC($\mathcal{M}$) qui est de traiter la résolution parallèle de problèmes de contraintes. Dans un tel contexte, la coupure dans les arbres de recherche n'est plus assurée par des gardes mais par des heuristiques plus évoluées et peu adaptées à un mécanisme de gardes[1]. Le langage CC($\mathcal{M}$) apparaît comme un successeur des systèmes non-déterministes plutôt que des langages gardés ainsi que la différence est faite dans [25, 26, 24] ou dans [109] entre Or-Parralel Prolog et FGHC.

[1]Par exemple, pour un point de choix donné, on s'attachera à évaluer en premier la branche qui maximise une certaine entropie [18, 17, 16]. Une telle heuristique est au mieux exprimée de manière maladroite avec des gardes, alors qu'elle est naturelle avec un algorithme de sélection de la branche à évaluer couplée avec une composition parallèle non-déterministe sans garde.

1.1.3. Une gestion du non-déterminisme aidée par le matériel

La gestion des références multiples dans le projet CC($\mathcal{M}$) est gérée en partie par le matériel par le biais du mécanisme de la pagination mémoire. Cette gestion est un hybride de tous les mécanismes existants de la gestion des références multiples pour le non-déterminisme.

En effet, on est proche des modèles basés sur des ordinateurs à mémoire non-partagée [2] (où la gestion des références multiples est complètement prise en charge par le matériel) à la différence près que le coût de création d'un agent n'est plus aussi élevé (il suffit de copier la mémoire locale et non plus toute la mémoire).

Le modèle de CC($\mathcal{M}$) est aussi proche de la philosophie des modèles avec copie de &ACE [84, 86], Muse [4], mais la copie est remplacée par une gestion des pages mémoires.

On peut aussi voir le modèle de mémoire à deux niveaux comme une version radicale de la copie sélective des environnements clos de [38].

Enfin, on peut aussi rapprocher, du fait de la localisation des variables locales sur chaque agent, le modèle de CC($\mathcal{M}$) des modèles utilisant des tables de hachage locales à chaque agent OU pour stocker les adresses locales des variables. On trouve ce type d'implantation dans les travaux de Ciepielewski et Haridi [27, 28], le modèle d'Argonne [11], le système PEPSys [5], AKL [64] et Oz [102, 104].

Le modèle CC($\mathcal{M}$) est toujours au moins aussi rapide que tous les modèles ainsi présentés.

1.2. Comparaison des modèles d'exécutions

1.2.1. Les modèles des langages logiques parallèles

Le modèle d'exécution de langage CC($\mathcal{M}$) est proche du modèle ET-OU proposé par [37] en ce sens où le langage CC($\mathcal{M}$) propose deux types de processus. Les premiers partagent leur mémoire locale et implantent le parallélisme-et. Les seconds ne le partagent pas et implantent le parallélisme-ou.

Le modèle de mémoire est lui un croisement entre le modèle SRI [114, 115] et le modèle de &ACE [84, 86] ou Muse [4]. En effet, du fait que chaque agent possède son espace mémoire local, on peut comparer ce segment propre aux tables de références du modèle SRI. D'un autre côté, l'installation de chaque segment de mémoire locale, installation effectuée au début de la fenêtre de temps allouée à chaque agent, rappelle le mécanisme de copie de &Ace et de Muse.

1.2.2. CC($\mathcal{M}$) et le modèle d'Andorra

La comparaison entre CC($\mathcal{M}$) et le modèle Andorra [10] que l'on peut trouver par exemple dans AKL est très instructive. La première implantation du langage CC($\mathcal{M}$)était orthogonale au principe du modèle d'Andorra, qui privilégie les réductions déterministes par rapport aux réductions non-déterministes. Elle s'est vite trouvée en défaut devant l'explosion du nombre d'agents actifs dans un contexte de ressources limitées. Pour remédier à ce problème, il a été implanté un mécanisme d'allocation de ressources lors de la création d'un agent. Celui-ci spécifie combien d'autres agents non-déterministes il est capable de créer. Ceci est fait grâce à un argument optionnel à l'instruction `detach local <entier> f()`. Un agent ainsi créé se retrouve bloqué tant qu'il ne peut réserver les ressources nécessaires en segments de mémoire locaux. Par ce biais, on privilégie les réductions déterministes par rapport aux

réductions non-déterministes. Le langage implante donc une version bornée du principe d'Andorra qui autorise les réductions non-déterministes tant que leur nombre ne dépasse pas une certaine borne.

Mais cette borne sur le nombre d'agents non-déterministes actifs joue en défaveur du langage CC($\mathcal{M}$). Il faudra donc dans le futur développer un schéma mixte d'implantation du non-déterministe à base de retour en arrière (¡¡backtrack¿¿) et de segments de mémoire locaux. Ce schéma peut s'inspirer de la double technique d'implantation du non-déterminisme dans [57].

1.2.3. CC($\mathcal{M}$) et Oz

Le modèle d'exécution du langage Oz est très proche de celui du langage CC($\mathcal{M}$). En effet, les deux proposent une recherche encapsulée avec une base de faits qui se partitionne (¡¡store splitting¿¿). La différence vient donc du modèle de mémoire. Au début de la conception du langage CC($\mathcal{M}$), il a été décidé de ne pas implanter un mécanisme de gardes profondes ou plates. De ce fait, la gestion mémoire est plus simple dans CC($\mathcal{M}$) que dans Oz. En effet, dans Oz, l'évaluation d'une garde profonde peut instancier des variables locales qui sont par la suite rajoutées à la base de faits quand la garde est promue (voir [102]). Ce mécanisme de réunion de deux bases de faits locales est implanté par des tables de hachage locales[71] qui ont l'inconvénient d'induire un sur-coût très important dans l'accès aux variables.

Donc, si les modèles d'exécution de CC($\mathcal{M}$) et de Oz sont très proches, la plus grande simplicité de celui de CC($\mathcal{M}$), due en particulier à l'opérateur $\oplus$ simplifie la gestion mémoire et permet donc une implantation plus optimisée.

Mais on peut établir une comparaison plus intéressante entre un sous-langage de Oz nommé Plain [74] et CC($\mathcal{M}$). En effet, ce langage reprend certaines des idées fondatrices de Oz et en particulier n'introduisent pas une base de faits basée sur l'unification. Le résultat est un langage impératif objet très proche de CC($\mathcal{M}$). Les variables logiques deviennent alors des canaux de communication et sont à la base de la synchronisation entre les processus comme dans le langage CC($\mathcal{M}$).

2. Bilan et perspectives

2.1. Du parallélisme à la distribution

On peut résumer l'apport de cette thèse du point de vue pratique en un point : le langage CC($\mathcal{M}$) permet la première implantation parallèle des domaines finis. De plus, grâce à des techniques évoluées d'implantation du parallélisme, des objets et du typage, cette implantation est efficace.

Mais l'effort d'implantation doit être poursuivi pour introduire la distribution. Cette extension apparaît nécessaire et difficile. Elle est nécessaire car c'est la seule possibilité pour augmenter la puissance que l'on peut allouer à un calcul donné. Mais cette extension pose plusieurs problèmes. Elle est coûteuse car les temps de communication entre machines distantes sont à plusieurs ordres de grandeur de ceux entre deux processeurs sur une même station de travail. Au delà même de ce coût, c'est la notion même de résolution de contraintes distribuées qui est floue. La théorie hésite entre les paradigmes par messages [90] et les paradigmes par mémoire virtuelle distribuée, ou leurs variantes avec des objets mobiles.

Mais nous sommes aidés dans cette tâche par deux faits importants. Tout d'abord, nous pouvons utiliser la décoration pour décrire de manière formelle un calcul distribué. En effet, nous avons vu comment il était possible avec la décoration de rajouter à un calcul une notion de date et de localisation. Ces deux informations peuvent nous servir à traduire la topologie d'un réseau de stations de travail et les temps de communication. On dispose alors d'un modèle formel de la distribution vu d'un point de vue très pragmatique, ce qui pourrait nous permettre d'étudier plus en avant la notion de résolution distribuée de problèmes de contraintes.

Enfin, la résolution de problèmes de contraintes s'effectuant de manière monotone, celle-ci peut se faire de manière asynchrone dans un environnement distribué. Cette amélioration simplifie énormément la distribution de la résolution de contraintes dans un paradigme par messages puisqu'elle ne nécessite pas de blocages globaux lors de l'ajout d'une contrainte à tous les sites distribués.

2.2. Vers d'autres utilisations de la décoration

La décoration est à notre avis la première contribution formelle pour décrire le contrôle de l'exécution d'un programme `CC`. Il reste quand même à développer le lien entre un contrôle d'exécution et une fonction de sélection sur une base de faits décorée. En effet, la théorie doit être complétée par la possibilité d'associer à tout algorithme de contrôle des `CC` un système de décoration adapté qui recode par une fonction de sélection le contrôle de départ.

Mais la décoration ouvre des perspectives plus générales et intéressantes. On peut utiliser la décoration afin de rendre compte d'une hiérarchie de contraintes [6] ou d'un ATMS [43]. En effet, la notion de sélection a posteriori peut être utilisée pour privilégier certaines contraintes par rapport à d'autres ou gérer des hypothèses. Par exemple, la décoration peut permettre à l'utilisateur de choisir certaines stratégies, de privilégier des contraintes, d'en effacer d'autres.

On peut aussi utiliser la décoration pour introduire les extensions réactives de la programmation logique par contraintes [49, 48]. Dans ce cas, elle permettrait de concevoir de nouveaux modèles d'exécution basés sur un contrôle dans les `CC`.

Bibliographie

[1] H. Aït-Kaci and A. Podelski. « Toward a meaning of life ». Technical Report, DEC PRL, June 1991.

[2] K. Ali. « Or-Parallel Execution of Prolog on BC-Machine ». In *Fifth International Conference of Logic Programming*, pages 1531–1545. MIT Press, 1988.

[3] K. Ali and R. Karlsson. « The Muse Or-Parallel Prolog Model and its Performance ». In *North American Conference on Logic Programming*. MIT Press, 1990.

[4] K. Ali and R. Karlsson. « Full Prolog and Scheduling Or-parallelism in Muse ». In *International Journal of Parallel Programming*, number 19 in 6, page 445 :475, 1991.

[5] U. baron, J. de Kergommeaux, H. Hailperin, M. Ratcliffe, P. Robert, and J. S. nad H. Westphal. « The Paraller ECRC Prolog System PEPSys : an Overview and Evaluation Results ». In *International Conference on Fith Generation Computer Systems*, pages 841–849, Tokyo, 1988. ICOT.

[6] A. B. B.F. Berson, J. Maloney. « An incremental constraint solver ». In CACM, editor, *Journal of Lisp and Functionnal Programming*, volume 33, January 1991.

[7] G. Birkhoff. *Lattice Theory*. American mathematical Society, 1963.

[8] P. Boizumault. « A General Model to Implement DIF and FREEZE ». In *ICLP*, pages 585–592, 1986.

[9] A. Bondarenko, F. Toni, and R. Kowalski. « An Assumption-Based Framework for Non-Monotonic Reasoning ». In A. Nerode and L. Pereira, editors, *Proceedings Second International Workshop on Logic Programming and Non-Monotonic Reasoning*. MIT Press, 1993.

[10] P. Brand, S. Haridi, and W. D.H.D.. « Andorra Prolog, the language and its application in distributed applications ». In *FGCS,88*, 1988.

[11] R. Butler, E. Lusk, R. Olson, and R. Overbeek. « ANLWAM : A Parallel Implementation of the Warren Abstract Machine ». Technical Report, MCS division, Argonne National Laboratory, 1986.

[12] L. Cardelli. « A semantics of multiple inheritance ». *Information and Computation*, 76 :138–164, 1988.

[13] B. Carlson, S. Haridi, and S. Janson. « AKL(FD), A concurrent language for FD programming ». In M. Bruynooghe, editor, *International Symposium on Logic Programming*, pages 521–536. MIT press, 1994.

[14] M. Carlsson. « freeze, Indexing and Others Implementation Issues in the Wam ». In *ICLP*, pages 40–58, May 1987.

[15] M. Carlsson. « *Design and Implementation of an OR-Parallel Prolog Engine* ». PhD thesis, SICS, 1990.

[16] Y. Caseau and F. Laburthe. « Improved CLP Scheduling with task intervals ». In P. V. Hentenryck, editor, *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, Santa Margherita Ligure, Italy, 1994. MIT Press.

[17] Y. Caseau and F. Laburthe. « Disjunctive Scheduling with Task Intervals ». Technical Report, LIENS Technical Report 95-25, École Normale Supérieure Paris, France, July 1995.

[18] Y. Caseau and F. Laburthe. « Improving Branch and Bound for Jobshop Scheduling with Constraint Propagation ». In M. Deza, R. Euler, and Y. Manoussakis, editors, *Combinatorics and Computer Science, 8th Franco-Japanese 4th Franco-Chinese Conference*, to appear in LNCS, Brest, France, 1996. Springer Verlag.

[19] Y. Caseau and L. Perron. « A type system for object-oriented database programming and querying languages ». In *Proc. of Workshop on Database Programming Languages*, Grece, 1991.

[20] Y. Caseau and L. Perron. « Attaching second-order types to methods in an object-oriented l anguage ». In *Proc. of ECOOP'93*, Kaiserslautern, 1993.

[21] Y. Caseau and F. Laburthe. « *Claire, reference manual* ». Ecole Normale Suprieure.

[22] G. Castagna. *Objetc-Oriented Programming, A Unified Foundation*. Borkhäuser, 1997.

[23] K. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[24] J. Chassin de kergommeaux and P. Codognet. « Parallel Logic Programming Systems ». *ACM Computing Survey*, 26(3) :295–336, september 1994.

[25] J. Chassin de Kergommeaux, P. Codognet, P. Robert, and J.-C. Syre. « Une programmation logique parallle : Langages gards ». *TSI*, 03 :205–224, 1989.

[26] J. Chassin de Kergommeaux, P. Codognet, P. Robert, and J.-C. Syre. « Une programmation logique parallle : Systmes non-dterministes ». *TSI*, 04 :285–305, 1989.

[27] A. Ciepielewski and S. Haridi. « Storage Models for Or-Parallel Execution of Logic Programs ». Technical Report, The Royal Institute of Technology, Stockholm, 1983.

[28] A. Ciepielewski, S. Haridi, and B. Hausman. « OR-Parallel Prolog on Shared Memory Multiprocessors ». *Journal of Logic Programming*, 7 :125 :147, 1989.

[29] K. Clark and S. Gregory. « PARLOG, Parallel Programming in Logic ». *ACM transaction on Programming Languages Systems*, 8(1) :1–49, 1986.

[30] E. V. d. l. Clergerie and B. Lang. « LPDA : another look at Tabulation in logic programming ». In *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 470–486, 1994.

[31] P. Codognet, F. Fages, and T. Sola. « A meta-level compiler for CLP(FD) and its combination with intelligent backtracking ». In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming : Selected Research*. MIT Press, 1993.

[32] P. Codognet and D. Diaz. « clp(B) : Combining Simplicity and Efficiency in Boolean Constraint Solving ». In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, pages 244–260. Springer Verlag, 1994.

[33] P. Codognet and D. Diaz. « A minimal extension of the WAM for `clp(FD)` ». In *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, June 1994. MIT Press.

[34] P. Codognet and V. A. Saraswat. « Abduction in Concurrent Constraint Programming ». Technical Report, INRIA-Rocquencourt, March 1992. (Based on abstract presented at Compulog workshop "AI & LP").

[35] A. Colmerauer. « PROLOG, Language of Artificial Intelligence ». *RECHERCHE (FRANCE) ISSN : 0029-5671*, 15(158) :1104–14, September 1984.

[36] A. Colmerauer. « An Introduction to Prolog III ». *Communications of the ACM*, 33(7) :69–90, July 1990.

[37] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, 1987.

[38] J. Conery. « Binding Environments for Parallel Logic Programs in Non-shared Memory Multiprocessors ». In *SLP*, pages 457–467, 1987.

[39] P. Cousot. « Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice ». Technical Report, Laboratoire IMAG, Universit scientifique et mdicale de Grenoble, Grenonle, FRANCE, 1988.

[40] P. Cousot and R. Cousot. « Abstract interpretation, A Unified Lattice Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints ». In *Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[41] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. « Non-monotonic Concurrent Constraint Programming ». In *ILPS : Proceedings 3rd International Logic Programming Symposium*, Vancouver, 1993.

[42] F. S. de Boer, A. Di Pierro, and C. Palamidessi. « An algebraic perspective of constraint logic programming ». *Journal Of Logic Computation*, 7(1) :1–38, 1997.

[43] J. De Kleer. « An Assumption-based TMS ». *Artificial Intelligence*, 28(2) :127–224, March 1986.

[44] M. Dincbas. « The CHIP Constraint Programming System ». In *5th Israeli Symposium on Artificial Intelligence*, Tel Aviv, Israel, December 1988. (invited talk).

[45] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. « The Constraint Logic Programming Language CHIP ». In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988.

[46] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and A. Herold. « The CHIP System : Constraint Handling in Prolog ». In L. E. and R. Overbeek, editors, *9th International Conference on Automated Deduction*, Argonne, Ill, May 1988. Springer.

[47] F. Fages. *Programmation Logique par Contraintes.* Collection Cours de l'Ecole Polytechnique. Ed. Ellipses, Paris, 1996.

[48] F. Fages, J. Fowler, and T. Sola. « A Reactive Constraint Logic Programming Scheme ». In L. Sterling, editor, *International Conference on Logic Programming, ICLP*, Tokyo, 1995. MIT Press.

[49] F. Fages, J. Fowler, and T. Sola. « Experiments in Reactive Constraint Logic Programming ». *To appear in the Journal of Logic Programming*, 1996.

[50] F. Fages and T. Sola. « Delay mechanisms with priorities for constraint solving ». In *Proc. of the first Workshop on Constraint Logic Programming, WCLP'91*, Marseille, 1991.

[51] A. Goldberg and D. Robson. *Smalltalk 80, the langage and its implementation.* Addison-Wesley, 1983.

[52] L. Henkin, J. Monk, and A. Tarski. *Cylindric algebras.* North-Holland, 1971.

[53] L. Henkin, J. Monk, and A. Tarski. *Cylindric algebras, Part II.* North-Holland, 1985.

[54] P. V. Hentenryck. « Parallel Constraint Satisfaction in Logic Programming : Preliminary Results of CHIP within PEPSys ». In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[55] P. V. Hentenryck. « The CLP Language CHIP : Constraint Solving and Applications ». In *COMPCON-91*, San Francisco, CA, February 1991.

[56] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1 edition, 1989.

[57] M. Hermenegildo, M. Cabeza, and M. Carro. « Using attributed variables in the implementation of Concurrent and Parallel Logic Programming Systems ». In L. Sterling, editor, *ILPS*, pages 631–645, 1995.

[58] C. Holzbaur. « Metastructures vs. Attributed Variables in the Context of extensible Unitication ». In *PLILP*, pages 260–268, 1992.

[59] C. Holzbaur. « A High-Level Approach to the Realization of CLP Languages ». In *Proceedings of the JICSLP92 Post-Conference Workshop on Constraint Logic Programming Systems*, pages 260–268, Washington D.C, 1992.

[60] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. « The CLP($\mathcal{R}$) Language and System ». *ACM transaction on Programming Langauges*, 14(3) :339–395, 1992.

[61] J. Jaffar and J. L. Lassez. « Constraint Logic programming ». Technical Report, IBM, 1986.

[62] J. Jaffar and J.-L. Lassez. « Constraint Logic Programming ». In *POPL'87 : Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987. ACM.

[63] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. « Angelic non-determinism in concurrent constraint programming ». Technical Report, System Sciences Laboratory, Xerox PARC, January 1991.

[64] S. Janson. « *AKL, A Multiparadigm Programming Language* ». PhD thesis, SICS, Uppsala University, 1994.

[65] J. Jourdan. « *Concurrence et coopration de modles multiples dans les langages CLP et CC : vers une mthodologie de programmation par modlisation* ». PhD thesis, Universit de Paris VII, 1995.

[66] T. Keisu. « *Tree Constraints* ». PhD thesis, SICS, Uppsala University, 1994.

[67] C. Laneve and U. Montanari. « Mobility in the CC Paradigm ». In *Proceedings of Mathematical Foundations of Computer Science*, 1992.

[68] S. Le Houitouze. « A New Data Structure for Implementing Extensions to Prolog ». In *PLILP*, pages 136–150, 1990.

[69] X. Leroy. « *Objective Caml Reference Manual* ». INRIA, 1996.

[70] M. Maher. « Logic Semantics for a Class of Committed-Choice Programs ». In J.-L. Lassez, editor, *ICLP : Proceedings 4th International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987. MIT.

[71] M. Mehl, R. Scheidhauer, and C. Schulte. « An Abstract Machine for Oz ». Research Report RR-95-08, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D66123 Saarbrücken, Germany, June 1995. Also in : *Proceedings of PLILP'95*, Springer-Verlag, LNCS, Utrecht, The Netherlands.

[72] U. Montanari, F. Rossi, and V. Saraswat. « CC Programs with both In- and Nondeterminism, a concurrent Semantics ». Technical Report, Acclaim, 1995.

[73] M. Müller, T. Müller, and P. Van Roy. Multi-Paradigm Programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Visions for the Future of Logic Programming : Laying the Foundations for a Modern successor of Prolog*. Portland, Oregon, 7 December 1995. A Workshop in Association with ILPS'95.

[74] M. Müller, J. Niehren, and G. Smolka. « Typed Concurrent Programming with Logic Variables ». Technical Report, Programming Systems Lab, Universität des Saarlandes, 1997. Submitted.

[75] T. Müller. « Adding Constraint Systems to DFKI Oz ». In *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 29 November–1 December 1995.

[76] T. Müller and J. Würtz. « Interfacing Propagators with a Concurrent Constraint Language ». In *JICSLP96 Post-conference workshop and Compulog Net Meeting on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 195–206, 1996.

[77] M. Nilsson and T. Hidehiko. « A Flat GHC implementation for Supercomputers ». In *Fifth IJSLP*, pages 1337–1350. MIT Press, 1988.

[78] L. Perron. « cc($\mathcal{M}$), a kernel for implementing cc languages ». In *Proc. of CCP'95*, 1995.

[79] L. Perron. « A concurrent constraint kernel language based on messages ». In *Proc. of ICLP*, 1995. Poster.

[80] L. Perron. « An implementation of or-parallelism based on direct access to the MMU ». In *Proc. of Compulog-Net worshop on parallelism and implementation technology, JICSLP'96*, 1996.

[81] G. Pesant and M. Gendreau. « A View of Local Search in Constraint Programming ». In *Principles and Practice of Constraint Programming - CP96 : Proceedings of the Second International Conference*, volume 1118 of *Lecture Notes in Computer Science*, pages 353–366, Berlin, 1996. Springer-Verlag.

[82] G. Plotkin. « A Structured Approach to Operational Semantics ». Technical Report FN-19, Computer Science Department, Aarhus University, 1981.

[83] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems. A survey of current trends. In *Current Trends in Concurrency, Overviews and Tutorial*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.

[84] E. Pontelli, G. Gupta, and M. Hermenegildo. « &ACE :A high performance parallel prolog system ». In *IIPS 95*, Santa Barbara, CA, 1995. IEEE Computer Society.

[85] E. Pontelli. « Adventure in Parallel Logic Programming ». http ://www.cs.nmsu.edu/ epontell/adventure/.

[86] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE, The And-Parallel Component of ACE (A Progess Report on ACE). In J. Barlklund, B. Jayaraman, and J. Tanaka, editors, *Parallel and Data Parallel Execution of Logic Programming*, volume 78, pages 65–78. UPMAIL, June 1994.

[87] POSIX. « 1003.4a threads extentions ».

[88] S. Prestwich and S. Mudambi. « Cost-Parallel Branch-and-Bound in Constraint Logic Programming ». In *ILPS Workshop on Constraint Languages and their use in Problem Modelling*, 1994.

[89] W. Rounds and G.-Q. Zhang. « Constraints in Non-Monotonic Reasoning ». In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *PPCP'93 : First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[90] J.-H. Rty. « *Langages Concurernts avec Contraintes : Communication par Messages et Distribution* ». PhD thesis, Universit d'Orleans, 1997.

[91] V. Saraswat. « Concurrent Constraint Programming ». In *seventeenth ACM Symposium on Principles of Programming Languages*, 1990.

[92] V. Saraswat. « The Category of Constraint Systems is Cartesian-Closed ». In *LICS*, pages 341–345, 1992.

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

[94] V. Saraswat, M. Rinard, and M. Panangaden. « Semantic foundations of Concurrent Constraint Programming ». In *eightteenth ACM Symposium on Principles of Programming Languages*, 1991.

[95] C. Schulte and G. Smolka. « Encapsulated Search in Higher-order Concurrent Constraint Programming ». In M. Bruynooghe, editor, *Logic Programming : Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, November 1994. MIT-Press.

[96] C. Schulte, G. Smolka, and J. Würtz. « Encapsulated Search and Constraint Programming in Oz ». In A. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[97] D. S. Scott. « domain for denotationals semantics ». In *Proceedings of ICALP*, 1982.

[98] E. Shapiro. « An or-Parallel Execution Algorithm for Prolog and its FCP implementation ». In *Fourth International Comference of Logic Programming*, pages 311–317. MIT Press, 1987.

[99] K. Shen. « *Studies in And/Or Parallelism in Prolog* ». PhD thesis, University of Cambridge, 1992.

[100] SICS. « *SICStus Prolog User's Manual* », June 1995.

[101] D. A. Smith. « MultiLog : Data Or-Parallel Logic Programming ». In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 314–331, Budapest, Hungary, 1993. The MIT Press.

[102] G. Smolka. « A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards ». Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, 'D-66123 Saarbrücken, Germany, February 1994.

[103] G. Smolka. « A Foundation for Higher-order Concurrent Constraint Programming ». In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 September 1994. Springer-Verlag.

[104] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[105] G. Smolka, M. Henz, and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT Press, 1994.

[106] G. Smolka and R. Treinen. « Records for Logic Programming ». *Journal of Logic Programming*, 18(3) :229–258, April 1994.

[107] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[108] Sun Corporation. « *The Java Programming Language* », 1996.

[109] E. Tick. *Parallel Logic Programming*. MIT Press, 1991.

[110] K. Ueda. « *Guarded Horn Clauses* ». PhD thesis, University of Tokyo, 1986.

[111] P. Van Hentenryck. « Parallel Constraint Satisfaction in Logic Programming : CHIP within PEPSys ». In G. Levi and M. Martelli, editors, *ICLP : Proceedings 6th International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.

[112] P. Van Hentenryck, V. Saraswat, and Y. Deville. « Constraint Processing in `cc(FD)` ». Technical Report, Brown University, 1992.

[113] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, Implementation, and Evaluation of the Constraint Language cc(FD). In A. Podelski, editor, *Constraint Programming : Basics and Trends*, LNCS 910. Springer, 1995. (Châtillon-sur-Seine Spring School, France, May 1994).

[114] D. Warren. « Or-Parallel Execution Models of Prolog ». In *TATSOFT'87*, pages 243,259. Springer Verlag, 1987.

[115] D. Warren. « The SRI Model for Or-Parallel Execution of Prolog : Abstract Design and Implementation ». In *SLP*, pages 92–102, 1987.

[116] N.-f. Zhou. « A Novel Implementation Method of Delay ». In M. Maher, editor, *JICSLP*, pages 97–111. MIT Press, 1996.

Table des matières