Mémoire de DEA

Gemplus Software Research Lab

Contrôle de ressource et évitement des interblocages sur la mémoire

Mathieu Baudet

Sous la direction de :

Jean Goubault-Larrecq — LSV - ENS Cachan
Antoine Galland — Gemplus Software Research Lab

Soutenu le 11 septembre 2002, pour l'obtention du
DEA Programmation : sémantique, preuves et langages
(Université Denis Diderot – Paris 7).

Cadre du stage

Mon stage de DEA s'est déroulé au sein du laboratoire de recherche en logiciels de la société Gemplus, situé à Gémenos(13). J'ai travaillé en collaboration avec Antoine Galland, qui réalise actuellement une thèse sur le contrôle de ressources dans les cartes à puce.

Remerciements

Je tiens à remercier Antoine Galland pour avoir rendu ce stage possible et pour la confiance qu'il m'a témoignée ; Jean Goubault-Larrecq pour le temps qu'il m'a consacré et pour ses précieux conseils ; Lilian Burdy et Antoine Requet pour la preuve formelle en B d'un lemme essentiel du chapitre 4. Je voudrais également remercier Laurent Lagosanto, Christophe Paris et toute l'équipe du laboratoire pour l'ambiance sympathique dans laquelle s'est déroulé mon stage.

Note

Cette version a fait l'objet de corrections typographiques mineures par rapport à la version originale. La définition 2.5.6 a également été complétée.

Résumé

Ce rapport présente les bases théoriques d'une architecture dédiée au contrôle de la ressource mémoire dans un système d'exploitation multi-processus. Par souci d'économie, on choisit de rendre les allocations impossibles bloquantes – plutôt que de déclencher une erreur – et de mettre en place une politique d'évitement des interblocages basée sur une analyse statique des programmes.

Dans le but de formaliser un critère de détection des états dangereux – i.e. des états conduisant forcément à un interblocage –, on introduit un petit langage de processus comportant les allocations et les libérations, un opérateur de séquence et un opérateur $\|$ pour l'exécution concurrente. Après avoir muni ce langage d'un pré-ordre sémantique naturel, on montre l'existence d'une traduction exacte des processus vers les listes – c'est-à-dire vers les processus purement séquentiels.

Cette traduction nous permet d'énoncer un critère de détection des états dangereux polynomial et exact ; elle nous incite également à étudier plus en détail l'ensemble pré-ordonné des listes. Au terme de l'étude, nous parvenons à en montrer la structure de treillis, ce qui constitue un résultat prometteur dans la perspective d'analyser des programmes réalistes, avec structures de contrôle.

mot-clés : mémoire, système d'allocation de ressources, évitement des interblocages, interprétation abstraite

Table des matières

1 Introduction **3**
1.1 Motivations . . . 3
1.2 Évitement des interblocages . . . 4
1.3 Travaux existants . . . 5
1.4 Contenu . . . 6

2 Étude d'une algèbre de processus **7**
2.1 Définition du langage . . . 7
2.1.1 Syntaxe et sémantique . . . 7
2.1.2 Contextes et transitions sur les contextes . . . 8
2.1.3 Bisimilarité forte . . . 8
2.1.4 Notations . . . 9
2.1.5 Traces d'exécution . . . 9
2.1.6 Coût et variation totale . . . 10
2.1.7 Critère de sûreté . . . 10
2.2 Algorithme par exploration des états . . . 10
2.3 Regroupement de variations . . . 11
2.3.1 Notion de bloc . . . 11
2.3.2 Redéfinition du langage . . . 11
2.3.3 Correspondance entre variations et blocs . . . 12
2.3.4 Règle de fusion . . . 12
2.4 Premier algorithme polynomial approché . . . 13
2.4.1 Principe . . . 13
2.4.2 Algorithme d'ordonnancement des blocs . . . 14
2.4.3 Généralisation aux listes triées . . . 16
2.4.4 Commentaires . . . 16
2.5 Algorithme polynomial exact . . . 16
2.5.1 Pré-ordre observationnel $\sqsubseteq$ . . . 17
2.5.2 Étude de la fusion, normalisation des listes . . . 18
2.5.3 Simplification de $\sqsubseteq$ . . . 22
2.5.4 Traduction de l'opérateur $\|$ . . . 23
2.5.5 Une traduction exacte vers les listes normalisées . . . 24
2.5.6 Retour aux variations . . . 24
2.5.7 Commentaires . . . 25

3 Application à l'ordonnancement de tâches **26**
3.1 Analyse statique des consommations 26
3.2 Phase dynamique : une librairie pour détecter les interblocages 27
3.3 Implémentation dans le cas séquentiel 28

4 Le treillis des listes **30**
4.1 Point de vue "absolu" 30
4.1.1 Définitions 30
4.1.2 Correspondance entre listes et vecteurs 31
4.2 Caractérisation de $\sqsubseteq$ sur les listes 32
4.2.1 Pré-ordre point par point modulo déformation du temps $\lesssim$ 32
4.2.2 Présentation de $\lesssim$ en termes de chemins 33
4.2.3 Équivalence de $\lesssim$ et $\sqsubseteq$ 34
4.3 Étude de $\sqsupseteq$ sur les listes normalisées 35
4.4 Existence et calcul de la borne supérieure $\sqcup$ 36
4.4.1 Étude de l'ensemble des majorants 36
4.4.2 Tests expérimentaux 37
4.4.3 Borne supérieure entre deux blocs 37
4.4.4 Calcul de $\sqcup$ entre deux listes triées 38

5 Conclusion et perspectives **40**
5.1 Conclusion 40
5.2 Travaux futurs 40
5.2.1 Branchements 40
5.2.2 Boucles 41
5.2.3 Non-terminaison 41
5.2.4 Un prototype plus réaliste 42
5.3 Limites de la démarche 42

A Exemples et résultats **43**
A.1 Exemples de calcul de coût 43
A.1.1 Algorithme par exploration des états 43
A.1.2 Algorithme 1 44
A.1.3 Algorithme 2 44
A.2 Étude expérimentale de la normalisation 45
A.3 Exemples d'exécution du prototype 45
A.3.1 Cas séquentiel 46
A.3.2 Petit langage impératif avec branchements et boucles 47

B Compléments **50**

C Énoncé du lemme 4.4.2 en B **51**

Chapitre 1

Introduction

1.1 Motivations

Contrairement aux plates-formes de calcul traditionnelles, les systèmes embarqués disposent souvent de peu de mémoire vive. Les cartes à puce[1] par exemple, se contentent généralement de 1 à 4 kilo-octets de RAM (mémoire rapide mais volatile) et de 16 à 128 kilo-octets d'EEPROM et/ou de FlashRAM (mémoires persistentes mais très lentes en écriture)[2].

Or, dans de nombreux secteurs, l'évolution rapide des marchés tend à prilégier les systèmes évolutifs, capables d'accueillir de nouvelles applications après leur mise en service. De tels systèmes, couplés à une plate-forme de développement efficace, permettent en effet de réduire le temps d'arrivée sur le marché des dernières fonctionnalités ce qui constitue un avantage concurrentiel important. Ainsi, dans le domaine des cartes à puce, sont apparues il y a quelques années des cartes dites "*ouvertes*", de type *Javacard* [Sun02] ou *Multos* [Mao02]. On peut citer également le système expérimental *Camille* [Gri00].

Dans un tel contexte, la gestion des ressources mémoire constitue un problème d'autant plus délicat. Sans mécanisme approprié, il y a en effet des risques de voir une application nouvellement installée – ou non – s'accaparer les ressources mémoire au détriment des autres, que ce soit par erreur ou par une attaque délibérée de type "déni de service".

Une première approche pour se prémunir de tels comportements est le *contrôle dynamique*. C'est la solution adoptée par [CvE98] dans *JRes*, une bibliothèque de composants permettant de surveiller l'activité d'un ensemble de threads Java (consommation mémoire, occupation du CPU, ...). Les avantages de cette démarche sont manifestement sa simplicité et sa généralité. Néanmoins lorsqu'une application "se conduit mal" (par exemple en demandant trop de mémoire), il est souvent trop tard pour réagir.

Une seconde approche – celle que nous adoptons ici – consiste à analyser statiquement les applications pour tenter de borner leurs besoins en mémoire. En cas de succès, on dispose alors d'une information *à l'avance* sur le comportement du programme, ce qui permet par exemple de refuser son installation si l'on détecte qu'il n'y a (peut-être) pas assez de mémoire.

[1] ou plus précisément les *cartes à micro-processeur*

[2] Le système d'exploitation et les bibliothèques de base sont quant à eux stockés dans une ROM de 32 à 128ko.

Le premier critère qui vient à l'esprit pour détecter le cas d'une mémoire totale insuffisante est simplement d'ajouter les consommations prévisionnelles maximales de chaque application et de comparer cette somme à la mémoire disponible. Néanmoins comme le remarque [GDGF02], une application est rarement à son maximum de consommation au cours de son exécution, de sorte qu'un tel critère peut conduire à un gâchis assez important.

Partant de ce constat, [GDGF02] propose de tirer parti de l'analyse pour piloter l'ordonnanceur de tâches, en vue cette fois *d'économiser de la mémoire*. L'idée est en effet qu'un ordonnanceur "bien renseigné" pourrait veiller à ce que les "pics" de consommation ne coïncident pas, ou encore pourrait faire passer une tâche qui va libérer de la mémoire en priorité.

Dans cet optique, nous allons maintenant chercher un algorithme efficace pour piloter l'ordonnanceur de tâches d'après des informations de consommation obtenues par une analyse statique.

1.2 Évitement des interblocages

Si l'on réduit la mémoire d'un système traditionnel, certaines allocations risquent de ne pas pouvoir être satisfaites, ce qui provoque généralement un message d'erreur et l'arrêt des programmes incriminés. Ce comportement, logique dans un système monotâche, ne paraît pas très adapté dans un environnement concurrent. En effet, même si une allocation est impossible à un moment donné, elle peut devenir réalisable plus tard, pour peu que le programme accepte de patienter. Désormais on s'intéresse donc à une architecture dans laquelle une allocation insatisfiable suspend l'exécution du processus en cours.

Avec les *allocations bloquantes* apparaissent malheureusement deux problèmes classiques des systèmes concurrents :

- la *famine*, situation dans laquelle un ou plusieurs processus sont bloqués indéfiniment, tandis que les autres continuent de s'exécuter.
- l'*interblocage* (ou *deadlock*), qui intervient lorsque tous les processus sont mis en attente faute de ressource.

Le problème de la famine peut être éludé si l'on considère des processus qui terminent en un temps fini, hypothèse que nous supposerons désormais satisfaite. Il reste à trouver un moyen de traiter le problème des interblocages. Traditionnellement on distingue trois approches possibles :

1. la *prévention* consiste à introduire des contraintes *a priori* sur le système dans le but de rendre les deadlocks impossibles. Par exemple, on peut exiger que les allocations soient faites en une seule fois au début de chaque programme et pour toute sa durée[3].
2. la *détection & résolution* consiste à détecter *a posteriori* les deadlocks, et à y remédier par un moyen approprié.
3. l'*évitement* nécessite des informations sur les consommations futures. Il consiste à mettre en oeuvre un algorithme spécifique pour détecter et éviter les *états dangereux* du système (nous verrons plus bas le sens à donner à cette expression).

La prévention conduit souvent à un gaspillage important des ressources. Nous ne nous y intéresserons donc pas. La détection & résolution suppose un moyen de "guérir" les deadlocks. Habituellement cela consiste à terminer certaines tâches, ou à revenir en arrière dans leur exécution. Dans le cas de la mémoire, on pourrait aussi imaginer un

[3] C'est d'ailleurs ce que préconise le standard actuel Javacard 2.2

mécanisme de *swapping*, capable de transférer temporairement les données d'un ou plusieurs programmes dans une mémoire auxiliaire pour soulager la mémoire principale (mais que faire lorsque cette mémoire auxiliaire est pleine ?).

Dans la suite nous nous intéressons plus particulièrement à l'évitement des interblogages, ce qui correspond en fait à la démarche annoncée au paragraphe précédent, à savoir un pilotage intelligent de l'ordonnanceur à partir d'informations statiques.

En utilisant la représentation géométrique des *graphes d'avancement* ("*progress graph*") attribuée à Dijkstra, on peut considérer un ordonnancement comme un chemin évitant des *zones interdites* – celles par définition où la somme des consommations instantanées des processus excède la mémoire totale disponible. Un état est dit *sûr* s'il existe un ordonnancement à partir de cet état qui permette de terminer toutes les tâches en cours. A l'inverse, les états non-sûrs (ou *dangereux*) conduisent nécessairement à un deadlock.

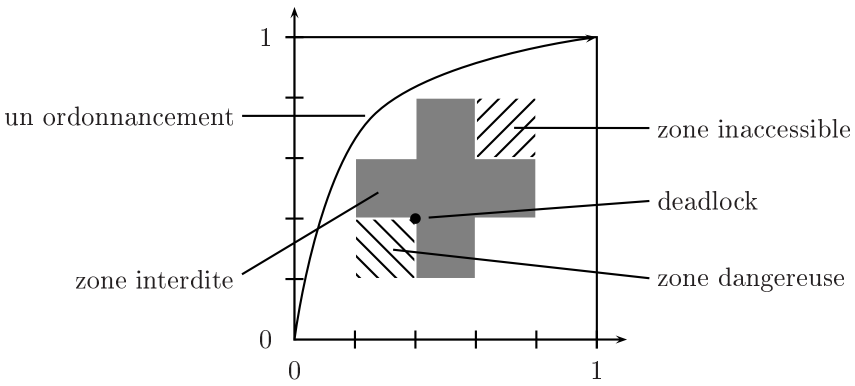


Fig. 1.1 – Un graphe d'avancement

Une stratégie d'évitement de deadlock consiste à vérifier avant chaque étape d'exécution que le système reste dans un état sûr. Dans le cas contraire, on met en attente le processus en cours, et on active le processus suivant dans la liste des tâches.

Pour appliquer une telle stratégie, il reste à trouver un critère effectif pour garantir *la sûreté de l'état courant*. C'est désormais ce à quoi nous allons nous employer en gardant à l'esprit que pour être applicable à un système embarqué, l'algorithme devra être à la fois efficace et économe en mémoire.

1.3 Travaux existants

Bien que le point de vue adopté dans la suite ne soit pas véritablement géométrique, je me suis inspiré au départ des travaux d'Éric Goubault sur l'approche géométrique de la théorie de la concurrence (voir par exemple [Gou00] et [Gou95]). Dans [FGR98], L. Fajstrup, E. Goubault et M. Raussen donnent un algorithme géométrique pour le calcul des régions dangereuses dans un système concurrent à plusieurs ressources. Cet algorithme résoud un problème beaucoup plus général que le nôtre au prix d'une complexité non polynomiale.

Le premier algorithme d'évitement de deadlock a été introduit par Dijkstra dans [Dij68] sous le nom d'*algorithme du banquier*. Généralisé par Haberman dans [Hab69], cette algorithme donne un critère de sûreté polynomial pour un système concurrent à plusieurs ressources renouvelables.

Depuis lors, l'évitement de deadlock a été étudié dans de nombreux systèmes d'allocation de ressource, et notamment pour des chaînes de fabrication industrielles (voir par exemple [LR01] et [RF96]).

Dans [Gol78], Gold montre la NP-complétude du problème dans le cas général, et donne une classification de la complexité de différents sous-problèmes. Le cas d'une seule ressource est déjà identifié comme étant polynomial. L'algorithme présenté au chapitre 2 peut être considéré comme une généralisation de l'algorithme de Gold à une algèbre de processus.

1.4 Contenu

Le chapitre 2 introduit un petit langage de processus comportant les allocations et les libérations de ressource, un opérateur de concaténation et un opérateur $\|$ pour l'exécution concurrente. On munit ce langage d'une sémantique par transitions, et d'un pré-ordre "observationnel" $\sqsubseteq$. Le résultat central du chapitre est une traduction sans perte d'information – au sens de $\sqsubseteq$ – des processus vers les listes, c'est-à-dire vers les processus sans opérateurs $\|$. Cette traduction donne lieu à un critère de sûreté exact de complexité polynomiale.

Le chapitre 3 applique les résultats du chapitre précédent à la mise en place d'une architecture permettant d'éviter les interblocages sur la mémoire. Après avoir décrit les deux parties statique et dynamique, on présente un prototype d'implémentation sur un langage purement séquentiel.

Le chapitre 4 entreprend une étude plus approfondie de l'algèbre de processus. On donne tout d'abord une caractérisation effective de $\sqsubseteq$. On montre ensuite que les listes normalisées constituent – comme on pouvait l'escompter – une famille de représentants des classes d'équivalence de $\sqsubseteq$. Pour finir, on met en évidence la structure de treillis des listes (et donc des processus).

On trouvera en annexe des exemples de calcul et les compléments de démonstrations.

Chapitre 2

Étude d'une algèbre de processus

2.1 Définition du langage

Afin d'étudier différents critères de sûreté, nous considérons un *langage de processus* minimaliste, dont le seul effet des programmes est d'allouer et de désallouer de la ressource. En dehors de cette ressource partagée, les processus n'ont aucune interaction mutuelle. Dans notre modèle, les programmes terminent toujours s'il y a suffisamment de mémoire. En revanche, on ne formule aucune hypothèse sur la ressource manipulée (consommable, récupérable...).

2.1.1 Syntaxe et sémantique

Les programmes[1] sont définis inductivement par :

$$\begin{array}{lcll} p & ::= & \epsilon & \text{processus vide} \\ & | & x & \text{variation } x \in \mathbb{Z} \text{ sur la ressource (allocation ou désallocation)} \\ & | & (p_1\, p_2) & \text{séquence} \\ & | & (p_1 \parallel p_2) & \text{exécution concurrente} \end{array}$$

On donne une sémantique "à petits pas" à l'aide de transitions étiquetées :

$$\frac{}{(\epsilon p) \xrightarrow{\epsilon} p} \qquad \frac{}{(\epsilon \parallel p) \xrightarrow{\epsilon} p} \qquad \frac{}{(p \parallel \epsilon) \xrightarrow{\epsilon} p} \qquad \frac{p \xrightarrow{\epsilon *} p'}{p \xrightarrow{\epsilon} p'} \qquad \text{(règles structurelles)}$$

$$\frac{}{x \xrightarrow{x} \epsilon} \qquad \frac{p \xrightarrow{\epsilon}\xrightarrow{x}\xrightarrow{\epsilon} p'}{E[p] \xrightarrow{x} E[p']} \qquad \text{(règles d'évaluation)}$$

où $E[\_]$ est un *contexte d'évaluation* défini par :

$$\begin{array}{lcll} E[\_] & ::= & \_ & \text{trou} \\ & | & (E[\_]\, p) & \text{séquence} \\ & | & (E[\_] \parallel p) & \text{parallèle gauche} \\ & | & (p \parallel E[\_]) & \text{parallèle droit} \end{array}$$

[1] Dans la suite, on emploiera indifféremment les mots programme, processus ou tâche pour désigner les termes du langage.

Une transition $\xrightarrow{x}$ consiste donc intuitivement en une série de transitions structurelles, suivie d'une allocation x en position d'évaluation, puis d'une autre série de transitions structurelles.

2.1.2 Contextes et transitions sur les contextes

L'ensemble des *contextes* est donné par :

$$\begin{array}{rcll} C[\_] & ::= & \_ & \text{trou} \\ & | & (C[\_]\,p) & \text{séquence gauche} \\ & | & (p\,C[\_]) & \text{séquence droite} \\ & | & (C[\_] \parallel p) & \text{parallèle gauche} \\ & | & (p \parallel C[\_]) & \text{parallèle droit} \end{array}$$

On convient de généraliser les transitions aux contextes (le trou n'intervenant pas dans les transitions).

Proposition 2.1.1. *Si* $C[\_] \xrightarrow{x} C'[\_]$, *alors :*

1. $\forall p,\ C[p] \xrightarrow{x} C'[p]$
2. *si* $C[\_]$ *est un contexte d'évaluation, alors* $C'[\_]$ *est un contexte d'évaluation*

Étant donné l'absence d'interaction entre processus concurrents, on dispose également de la propriété remarquable suivante :

Proposition 2.1.2. *Pour tous* $C[\_], p, q, x$,

$$C[p] \xrightarrow{x} q \Longrightarrow \ ou \left| \begin{array}{l} \exists C'[\_],\ C[\_] \xrightarrow{x} C'[\_] \text{ et } q = C'[p] \\ C[\_] \text{ est un contexte d'évaluation et } \exists p',\ p \xrightarrow{x} p' \text{ et } q = C[p'] \end{array} \right.$$

2.1.3 Bisimilarité forte

Avant d'aller plus loin, nous allons identifier les processus qui sont strictement équivalents du point de vue du système de transition $\xrightarrow{x}$. La technique – maintenant bien connue (voir par exemple [WN94]) – fait appel à la notion de *simulation*.

Définition 2.1.3. Soit $\mathcal{R}$ une relation.
– $\mathcal{R}$ est une *simulation forte* si :

$$\forall p_1, p_2, p_1', \forall x,\ (p_1 \mathcal{R} p_2 \text{ et } p_1 \xrightarrow{x} p_1') \Rightarrow (\exists p_2',\ p_2 \xrightarrow{x} p_2' \text{ et } p_1' \mathcal{R} p_2')$$

– $\mathcal{R}$ est une *bisimulation forte* si $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont des simulations fortes.
La plus grande des bisimulations fortes est appelée *bisimilarité forte* et notée $\approx^\circ$.

Fort de cette définition, on établit les propriétés suivantes (par coinduction) :

Proposition 2.1.4.

1. $\approx^\circ$ *est une relation d'équivalence*
2. $\approx^\circ$ *passe au contexte :* $\forall p_1, p_2, C[\_], \quad p_1 \approx^\circ p_2 \Rightarrow C[p_1] \approx^\circ C[p_2]$
3. $\forall p, \quad p \approx^\circ \epsilon p \approx^\circ p\,\epsilon \approx^\circ \epsilon \parallel p \approx^\circ p \parallel \epsilon$
4. $\forall p_1, p_2, p_3, \quad p_1(p_2 p_3) \approx^\circ (p_1 p_2) p_3$
5. $\forall p_1, p_2, \quad p_1 \parallel p_2 \approx^\circ p_2 \parallel p_1$
6. $\forall p_1, p_2, p_3, \quad p_1 \parallel (p_2 \parallel p_3) \approx^\circ (p_1 \parallel p_2) \parallel p_3$

2.1.4 Notations

Dans la suite, les programmes seront implicitement considérés modulo $\approx^\circ$, et l'on omettra les parenthèses inutiles. D'où par exemple :

$$p = (1(2 \parallel 3)\ -2) \parallel (2\ -3\ 4) \parallel 2$$

Les processus sans opérateur $\parallel$ seront appelés *liste de variations*. Modulo $\approx^\circ$, ce sont en fait des listes d'entiers au sens habituel.

On notera de plus :

- $p \underset{\neq}{\xrightarrow{l}} p'$ si $l = x_1 x_2 \dots x_n$ est une liste de variation, $n > 0$ et si $p \xrightarrow{x_1}\xrightarrow{x_2} \cdots \xrightarrow{x_n} p'$,
- $p \xrightarrow{l} p'$ si $p \underset{\neq}{\xrightarrow{l}} p'$ ou si $p \xrightarrow{\epsilon} p'$ et $l = \epsilon$,
- $p \underset{\neq}{\rightarrow} p'$ si $\exists l, p \underset{\neq}{\xrightarrow{l}} p'$
- et enfin $p \rightarrow p'$ lorsque $\exists l, p \xrightarrow{l} p'$

Proposition 2.1.5. *$\underset{\neq}{\rightarrow}$ est noétherienne. Le seul élément en forme normale (modulo $\approx^\circ$) est ϵ.*

En d'autres termes, cette dernière proposition nous dit que tous les programmes du langage terminent. On va maintenant étudier brièvement quelles sont les traces d'exécutions possibles d'un terme.

2.1.5 Traces d'exécution

Une *trace d'exécution* (ou *ordonnancement*) de p est une liste l telle que $p \xrightarrow{l} \epsilon$. L'ensemble des traces d'exécution de p est noté $\Lambda(p)$. D'après la définition de $\xrightarrow{l}$, il vérifie :

$$\begin{aligned} \Lambda(p) &= \bigcup_{p \xrightarrow{x} p'} x\Lambda(p') \quad \text{si } p \neq \epsilon \\ \Lambda(\epsilon) &= \{\epsilon\} \end{aligned}$$

Cette caractérisation permet de démontrer la propriété suivante par récurrence sur le nombre de variations de p :

Proposition 2.1.6. *Les traces de p sont des listes, obtenues par permutation des variations figurant dans p (en respectant le nombre d'occurences). En particulier, toutes les traces de p ont la même taille : le nombre de variations dans p ; et il y en a un nombre fini.*

Par exemple,

$$\Lambda(3\,(1\,2 \parallel 3)) = \{(3\,3\,1\,2),\ (3\,1\,3\,2),\ (3\,1\,2\,3)\}$$

Étant donné que les transitions sur les séquences sont déterministes, on peut noter également :

$$\Lambda(p_1\, p_2) = \Lambda(p_1)\,\Lambda(p_2) \overset{déf}{=} \{l_1\, l_2 | l_1 \in \Lambda(p_1), l_2 \in \Lambda(p_2)\}$$

2.1.6 Coût et variation totale

Après ces préliminaires, nous allons maintenant pouvoir définir le *coût* d'un processus, c'est-à-dire intuitivement la quantité de ressource nécessaire pour qu'il s'exécute entièrement ; ainsi que la *variation totale*, à savoir le résultat de toutes les variations une fois le processus terminé.

Pour ce faire, on définit tout d'abord le *coût* et la *variation totale* d'une liste :

$$\begin{aligned}
\mathcal{C}(x_1 x_2 \ldots x_n) &\stackrel{déf}{=} \max_{0 \leq i \leq n} \left(\sum_{1 \leq j \leq i} x_j \right) \quad (\geq 0) \\
\Delta(x_1 x_2 \ldots x_n) &\stackrel{déf}{=} \sum_{1 \leq i \leq n} x_i
\end{aligned}$$

(en particulier $\mathcal{C}(\epsilon) = \Delta(\epsilon) = 0$).

Le coût d'un programme se définit alors comme le coût minimum sur les traces d'exécution :

$$\mathcal{C}(p) \stackrel{déf}{=} \min_{p \xrightarrow{l} \epsilon} \mathcal{C}(l) \quad (\geq 0)$$

Enfin, si $p \xrightarrow{l} \epsilon$, alors la variation totale de p est par définition $\Delta(l)$, quantité qui ne dépend pas de de l d'après la propriété 2.1.6.

2.1.7 Critère de sûreté

Si l'on dispose d'un algorithme pour évaluer le coût d'un programme, on peut détecter très simplement si l'état courant est dangereux. Étant donné la mémoire totale disponible M et les différentes processus en cours $p_1, \ldots, p_N$, il suffit en effet de comparer M à $\mathcal{C}(p_1 \parallel p_2 \parallel \ldots \parallel p_N)$.

Désormais on va étudier différents algorithmes pour calculer le coût d'un programme (ou à défaut une majoration).

2.2 Algorithme par exploration des états

La définition du coût s'écrit également :

$$\begin{aligned}
\mathcal{C}(p) &= \min_{p \xrightarrow[\neq]{x} p'} \max(0, x + \mathcal{C}(p')), \quad \text{si } p \neq \epsilon \\
\mathcal{C}(\epsilon) &= 0
\end{aligned}$$

Un algorithme possible (issu de la programmation dynamique) consiste donc à parcourir le graphe des états intermédiaires (i.e. $G = \{q, p \xrightarrow{l} q\}$) à partir de l'état final, en mettant à jour successivement le coût des états. Cet algorithme permet de résoudre le problème en card(G) étapes. Malheureusement, G croît exponentiellement en la taille de p (phénomène d'*explosion combinatoire*), ce qui le rend inutilisable en pratique.

2.3 Regroupement de variations

2.3.1 Notion de bloc

Afin d'éviter l'écueil de l'explosion combinatoire, nous allons tenter de simplifier le problème par une *réécriture* des programmes à évaluer.

Plus précisément, dans le but de diminuer le nombre d'états du système, on peut avoir envie de regrouper certaines variations consécutives entre elles. Une manière naturelle de résumer une liste de variations l est simplement de prendre le couple $(\mathcal{C}(l), \Delta(l))$; on introduit donc la notion de bloc :

Définition 2.3.1. On appelle *bloc* tout couple $y = (c, \delta) \in \mathbb{N} \times \mathbb{Z}$ tel que $c \geq \delta$.

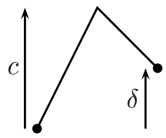


Fig. 2.1 – Un bloc (c, δ)

2.3.2 Redéfinition du langage

Toujours dans le but de faire apparaître les regroupements comme des opérations de réécriture, nous allons maintenant modifier notre langage de processus en remplaçant les variations par des blocs (y compris dans les transitions) :

$$
\begin{array}{rcll}
p & ::= & \epsilon & \text{processus vide} \\
 & | & (c, \delta) & \text{bloc } c \geq 0, c \geq \delta \\
 & | & (p_1 p_2) & \text{séquence} \\
 & | & (p_1 \parallel p_2) & \text{exécution concurrente}
\end{array}
$$

Les transitions deviennent :

$$
\frac{}{(\epsilon p) \xrightarrow{\epsilon} p} \qquad \frac{}{(\epsilon \parallel p) \xrightarrow{\epsilon} p} \qquad \frac{}{(p \parallel \epsilon) \xrightarrow{\epsilon} p} \qquad \frac{p \xrightarrow{\epsilon *} p'}{p \xrightarrow{\epsilon} p'} \qquad \text{(règles structurelles)}
$$

$$
\frac{}{(c, \delta) \xrightarrow{(c,\delta)} \epsilon} \qquad \frac{p \xrightarrow{\epsilon *} \xrightarrow{(c,\delta)} \xrightarrow{\epsilon *} p'}{E[p] \xrightarrow{(c,\delta)} E[p']} \qquad \text{(règles d'évaluation)}
$$

(avec la même définition de E)

On garde la même définition des contextes C ; les propositions 2.1.1 et 2.1.2 restent valables. La définition et les propriétés de $\approx^{\circ}$ se transposent également immédiatement ; on conserve donc les mêmes conventions d'écriture.

Le coût et la variation totale de processus à blocs sont définis comme précédemment :

$$
\begin{aligned}
\mathcal{C}(p) & \stackrel{déf}{=} \min_{p \xrightarrow{l} \epsilon} \mathcal{C}(l) \\
\Delta(p) & \stackrel{déf}{=} \Delta(l) \text{ si } p \xrightarrow{l} \epsilon
\end{aligned}
$$

avec cette fois :

$$\mathcal{C}\left((c_1,\delta_1)\dots(c_n,\delta_n)\right) \stackrel{déf}{=} \max_{1\leq i\leq n}\left(c_i + \sum_{1\leq j<i}\delta_j\right)$$

$$\Delta\left((c_1,\delta_1)\dots(c_n,\delta_n)\right) \stackrel{déf}{=} \sum_{1\leq i\leq n}\delta_i$$

2.3.3 Correspondance entre variations et blocs

On peut formaliser la correspondance entre les anciens et les nouveaux termes par la traduction suivante :

$$\begin{aligned} [\![\epsilon]\!]^0 &\stackrel{déf}{=} \epsilon \\ [\![x]\!]^0 &\stackrel{déf}{=} (\max(0,x),x) \\ [\![p_1p_2]\!]^0 &\stackrel{déf}{=} [\![p_1]\!]^0[\![p_2]\!]^0 \\ [\![p_1 \parallel p_2]\!]^0 &\stackrel{déf}{=} [\![p_1]\!]^0 \parallel [\![p_2]\!]^0 \end{aligned}$$

Une traduction en sens inverse pourrait être :

$$\begin{aligned} [\![\epsilon]\!]^{0'} &\stackrel{déf}{=} \epsilon \\ [\![(c,\delta)]\!]^{0'} &\stackrel{déf}{=} c\,(\delta - c) \\ [\![p_1p_2]\!]^{0'} &\stackrel{déf}{=} [\![p_1]\!]^{0'}[\![p_2]\!]^{0'} \\ [\![p_1 \parallel p_2]\!]^{0'} &\stackrel{déf}{=} [\![p_1]\!]^{0'} \parallel [\![p_2]\!]^{0'} \end{aligned}$$

L'équivalence entre les deux représentations sera formalisée plus loin. Mais on peut déjà noter :

Proposition 2.3.2. *Pour tous p, p', l,*

$$p \xrightarrow{l} p' \Leftrightarrow [\![p]\!]^0 \xrightarrow{[\![l]\!]^0} [\![p']\!]^0$$

$$\mathcal{C}(l) = \mathcal{C}([\![l]\!]^0) \text{ et } \Delta(l) = \Delta([\![l]\!]^0)$$

2.3.4 Règle de fusion

Maintenant que l'on dispose d'un langage plus "homogène", on peut formaliser la notion de regroupement de blocs (et donc de variations), par une règle de *fusion* :

$$(c_1,\delta_1)(c_2,\delta_2) \xrightarrow{fusion} (\max(c_1, c_2+\delta_1), \delta_1+\delta_2)$$

valable dans tout contexte $C[\_]$ (et définie modulo $\approx^\circ$).

Proposition 2.3.3. $\xrightarrow{fusion}$ *est fortement confluente et noethérienne. De plus, pour toutes listes l_1 et l_2, $l_1 \xrightarrow{fusion} l_2$ implique $\mathcal{C}(l_1) = \mathcal{C}(l_2)$ et $\Delta(l_1) = \Delta(l_2)$.*

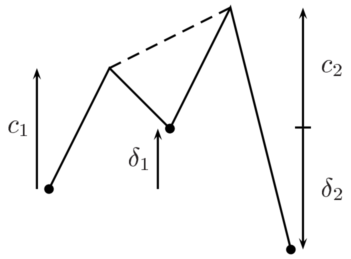


FIG. 2.2 – Fusion de deux blocs consécutifs

Démonstration. Étant donné que la règle *fusion* diminue la taille de la liste à chaque étape, elle est bien noethérienne. On montre la confluence forte en étudiant l'unique paire critique :

$$\begin{array}{ccc}
(c_1,\delta_1)(c_2,\delta_2)(c_3,\delta_3) & \longrightarrow & (\max(c_1,\delta_1+c_2),\delta_1+\delta_2)(c_3,\delta_3) \\
\downarrow & & \downarrow \\
(c_1,\delta_1)(\max(c_2,\delta_2+c_3),\delta_2+\delta_3) & \longrightarrow & (\max(c_1,\delta_1+c_2,\delta_1+\delta_2+c_3),\delta_1+\delta_2+\delta_3)
\end{array}$$

□

Remarque. La fusion complète d'une liste (non vide) s'écrit : $l \xrightarrow{fusion*} (\mathcal{C}(l),\Delta(l))$.

2.4 Premier algorithme polynomial approché

2.4.1 Principe

Une manière efficace de simplifier le problème consiste à ignorer les entrelacements entre processus. Formellement cela a revient à regrouper systématiquement toutes les variations consécutives, autrement dit à travailler avec un bloc par programme.

On définit donc une traduction qui à chaque programme p associe un bloc (c,δ), où c est une majoration du coût de p, et où δ est sa variation totale :

$$\begin{aligned}
[\![\epsilon]\!]^1 &\overset{déf}{=} (0,0) \\
[\![(c,\delta)]\!]^1 &\overset{déf}{=} (c,\delta) \\
[\![p_1p_2\dots p_n]\!]^1 &\overset{déf}{=} (\mathcal{C}\times\Delta)([\![p_1]\!]^1[\![p_2]\!]^1\dots[\![p_n]\!]^1) \\
[\![s_1 \parallel s_2 \parallel \dots \parallel s_n]\!]^1 &\overset{déf}{=} (\mathcal{C}\times\Delta)([\![s_1]\!]^1 \parallel [\![s_2]\!]^1 \parallel \dots \parallel [\![s_n]\!]^1)
\end{aligned}$$

où s_i désigne un processus qui *n'est pas* une parallélisation ($s := \epsilon|x|p_1p_2$).

Il reste à expliciter comment calculer :

$$\begin{array}{c}
\mathcal{C}((c_1,\delta_1) \parallel (c_2,\delta_2) \parallel \dots \parallel (c_n,\delta_n)) \\
\text{et}\ \ \Delta((c_1,\delta_1) \parallel (c_2,\delta_2) \parallel \dots \parallel (c_n,\delta_n))
\end{array}$$

Pour la seconde expression, le calcul est simple :

$$\Delta((c_1,\delta_1) \parallel (c_2,\delta_2) \parallel \dots \parallel (c_n,\delta_n)) = \sum_{1\le i\le n} \delta_i$$

La première expression en revanche relève d'un problème d'ordonnancement non trivial, qui par bonheur sera résolu par un algorithme de tri décrit dans le paragraphe suivant.

On pourrait démontrer dès maintenant la correction de cette traduction :

$$\forall p, \quad \mathcal{C}(p) \leq \mathcal{C}(\llbracket p \rrbracket^1)$$

mais on verra plus loin qu'elle découle directement de résultats plus généraux sur la fusion.

Remarque. La traduction $\llbracket \_ \rrbracket^1$ a été énoncée en regroupant les opérateurs $\|$ au même niveau, car elle est ainsi plus précise.

2.4.2 Algorithme d'ordonnancement des blocs

Étant donné n blocs (c_i, δ_i), il s'agit de calculer la quantité :

$$\mathcal{C}((c_1,\delta_1) \parallel (c_2,\delta_2) \parallel \ldots \parallel (c_n,\delta_n)) = \min_{\sigma \in \Sigma_n} \max_{1 \leq i \leq n} \left(c_{\sigma i} + \sum_{1 \leq j < i} \delta_{\sigma j} \right)$$

où Σ_n dénote l'ensemble des permutations sur $\{1 \ldots n\}$.

Dans la suite du paragraphe, nous allons montrer qu'il existe un algorithme de tri permettant de calculer directement un ordonnancement optimal. Le calcul du coût s'effectuera ainsi en un temps $O(n \log(n))$.

Définition 2.4.1. On définit la relation $\preceq^0$ sur les blocs par :

$$y_1 \preceq^0 y_2 \text{ ssi } \mathcal{C}(y_1 y_2) \leq \mathcal{C}(y_2\, y_1)$$

c'est-à-dire :

$$(c_1, \delta_1) \preceq^0 (c_2, \delta_2) \text{ ssi } \max(c_1, c_2 + \delta_1) \leq \max(c_2, c_1 + \delta_2)$$

On remarque que $\preceq^0$ est réflexive et totale. On a de plus la propriété suivante :

Lemme 2.4.2. *Soit $y_i = (c_i, \delta_i)$ des blocs $(i = 1, \ldots, n)$. Si pour tout i, $y_1 \preceq^0 y_i$, alors*

$$\mathcal{C}(y_1 \parallel \ldots \parallel y_n) = \mathcal{C}(y_1(y_2 \parallel \ldots \parallel y_n))$$

Démonstration. Le sens $\leq$ est évident (inclusion des chemins). Soit l un ordonnancement optimal pour $y_1 \parallel \ldots \parallel y_n$. l s'écrit : $l = y'_1\, y'_2\, \ldots\, y'_m\, y_1\, l'$. Comme $y_1 \preceq^0 y'_m$, la définition du coût implique :

$$\mathcal{C}(y'_1\, y'_2\, \ldots\, y'_m\, y_1\, l') \geq \mathcal{C}(y'_1\, y'_2\, \ldots\, y'_{m-1}\, y_1\, y'_m\, l')$$

En repétant l'opération $m - 1$ fois, on obtient :

$$\mathcal{C}(l) \geq \mathcal{C}(y_1\, y'_1\, y'_2\, \ldots\, y'_{m-1}\, y'_m\, l') \geq \mathcal{C}(y_1(y_2 \parallel \ldots \parallel y_n))$$

□

Étant donné que $\mathcal{C}(y_1(y_2 \parallel \ldots \parallel y_n)) = \max(c_1, \delta_1 + \mathcal{C}(y_2 \parallel \ldots \parallel y_n))$, ce lemme pourrait donner lieu à un raisonnement par récurrence, et nous fournir ainsi un algorithme de calcul du coût – sous réserve qu'il existe à chaque étape un y_{i_0} tel que $\forall i$ restant, $y_{i_0} \preceq^0 y_i$. Malheureusement l'existence d'un tel i_0 n'est pas assurée *a priori*, car $\preceq^0$ n'est pas transitive, comme le montre le contre-exemple suivant :

$(2,1) \preceq^0 (1,0)$ et $(1,0) \preceq^0 (2,-1)$ mais $(2,1) \not\preceq^0 (2,-1)$

Pour pallier ce défaut, on considère une nouvelle relation dont on va montrer qu'elle est réflexive, transitive, totale et incluse dans $\preceq^0$:

Définition 2.4.3. Soit $\preceq$ la relation sur les blocs définie par $(c_1, \delta_1) \preceq (c_2, \delta_2)$ ssi :

$$\begin{array}{r|ll} & \delta_1 \leq 0 \text{ et } \delta_2 \geq 0 & (1) \\ \text{ou} & \delta_1 < 0, \delta_2 < 0 \text{ et } c_1 \leq c_2 & (2) \\ \text{ou} & \delta_1 > 0, \delta_2 > 0 \text{ et } \delta_1 - c_1 \leq \delta_2 - c_2 & (3) \end{array}$$

Proposition 2.4.4. *$\preceq$ est un pré-ordre total.*

Démonstration.

1. réflexivité : clair.
2. transitivité :
   Supposons $(c_1, \delta_1) \preceq (c_2, \delta_2) \preceq (c_3, \delta_3)$.
   Si $\delta_1 < 0$, $\delta_2 < 0$ et $\delta_3 < 0$, on a : $c_1 \leq c_2 \leq c_3$.
   Si $\delta_1 > 0$, $\delta_2 > 0$ et $\delta_3 > 0$, alors : $\delta_1 - c_1 \leq \delta_2 - c_2 \leq \delta_3 - c_3$.
   Dans tous les autres cas, $\delta_1 \leq 0$ et $\delta_3 \geq 0$.
3. total :
   Soit (c_1, δ_1) et (c_2, δ_2) à comparer.
   Si δ_1 et δ_2 sont de signes différents, la règle (1) s'applique dans un sens ou dans l'autre.
   Sinon, l'une des règles (2) ou (3) s'applique. □

Proposition 2.4.5. *Pour tous blocs $(c_1, \delta_1), (c_2, \delta_2)$, on a :*

$$(c_1, \delta_1) \preceq (c_2, \delta_2) \Longrightarrow \max(c_1, c_2 + \delta_1) \leq \max(c_2, c_1 + \delta_2)$$

Démonstration. On raisonne par cas suivant $\preceq$:

1. Si $\delta_1 \leq 0$ et $\delta_2 \geq 0$, alors
$$\max(c_1, c_2 + \delta_1) \leq \max(c_1, c_2) \leq \max(c_2, c_1 + \delta_2)$$
2. Si $\delta_1 < 0$ et $\delta_2 < 0$ et $c_1 \leq c_2$, alors
$$\max(c_2, c_1 + \delta_2) = c_2 \geq \max(c_1, c_2 + \delta_1)$$
3. Si $\delta_1 > 0$ et $\delta_2 > 0$ et $\delta_1 - c_1 \leq \delta_2 - c_2$, alors
$$\max(c_2, c_1 + \delta_2) = c_1 + \delta_2 \geq \max(c_1, c_2 + \delta_1)$$

□

Compte tenu de ces résultats, on peut maintenant utiliser le lemme 2.4.2 pour montrer :

Théorème 2.4.6 (Ordonnancement des blocs). *Soit y_1, y_2, …, y_n des blocs. Si $y_1 \preceq y_2 \preceq \ldots \preceq y_n$, alors :*

$$\mathcal{C}(y_1 \parallel y_2 \parallel \ldots \parallel y_n) = \mathcal{C}(y_1\, y_2\, \ldots\, y_n)$$

Autrement dit, en notant $sort_{\preceq}$ un algorithme de tri quelconque selon $\preceq$, on a montré :

$$\mathcal{C}(y_1 \parallel y_2 \parallel \ldots \parallel y_n) = \mathcal{C}(sort_{\preceq}(y_1, y_2, \ldots, y_n))$$

2.4.3 Généralisation aux listes triées

On peut remarquer d'ores et déjà que l'algorithme d'ordonnancement des blocs se généralise immédiatement aux listes triées. En effet, une démonstration analogue à celle de 2.4.2 (mais utilisant en plus la transitivité de $\preceq$) permet de montrer :

Lemme 2.4.7. *Si les $l_i = y_i l'_i$ $(i = 1, \ldots, n)$ sont des listes* triées *pour $\preceq$, et si $y_1 \preceq y_i$ pour tout i alors*

$$\mathcal{C}(l_1 \parallel \ldots \parallel l_n) = \mathcal{C}(y_1(l'_1 \parallel l_2 \parallel \ldots \parallel l_n))$$

On en déduit alors par récurrence :

Théorème 2.4.8 (Ordonnancement des listes triées). *Soit l_1, l_2, …, l_n des listes de blocs triées pour $\preceq$. En notant $merge_{\preceq}(l_1, l_2, \ldots, l_n)$ la fusion des listes (au sens du tri-fusion), on a :*

$$\mathcal{C}(l_1 \parallel l_2 \parallel \ldots \parallel l_n) = \mathcal{C}(merge_{\preceq}(l_1, l_2, \ldots, l_n))$$

Remarque. Cette propriété est fausse en général sur des listes non triées, comme en témoigne le contre-exemple suivant :

$$\mathcal{C}(merge_{\preceq}((2,2)(0,-2),\ (1,1)(1,0))) = \mathcal{C}((1,1)(1,0)(2,2)(0,-2)) = 3$$

mais :

$$\mathcal{C}((2,2)(0,-2) \parallel (1,1)(1,0)) = \mathcal{C}((2,0) \parallel (2,1)) = 2$$

2.4.4 Commentaires

Une étude rapide de l'algorithme 1 montre que sa complexité en temps est en $O(n \log(m))$ où n est la taille du processus à évaluer (nombre de variations et de ϵ) et où m est le nombre de branchements maximales pour un nœud de type $\parallel$.

Cette algorithme efficace reste néanmoins très approché. En effet, comme les entrelacements sont ignorés, l'algorithme n'est –par exemple– pas capable de tirer partie d'un processus sur le point de désallouer de la mémoire[2].

2.5 Algorithme polynomial exact

On va maintenant chercher un algorithme qui calcule le coût exact d'un processus et non plus seulement une majoration.

Le défaut de l'algorithme 1 était visiblement de fusionner systématiquement les blocs consécutifs entre eux, ce qui entraînait la perte de certaines "informations utiles". Si nous voulons un algorithme exact, il nous faudra au contraire manipuler des listes de blocs. A cet égard, la généralisation de l'algorithme d'ordonnancement aux listes triées semble de bon augure.

[2] voir en annexe A pour un exemple de calcul

2.5.1 Pré-ordre observationnel $\sqsubseteq$

Dans le but d'étudier la règle de fusion ainsi que de prouver la correction des différentes traductions, on définit maintenant une relation de pré-ordre sémantique entre les termes.

Étant donné les éléments du langage, un choix naturel est d'utiliser la plus grande relation stable par contexte et vérifiant $p_1 \sqsubseteq p_2 \Rightarrow \mathcal{C}(p_1) \leq \mathcal{C}(p_2)$.

Définition 2.5.1 (Pré-ordre observationnel). On note $\sqsubseteq$ le pré-ordre défini par :

$$p_1 \sqsubseteq p_2 \text{ ssi } \forall C,\ \mathcal{C}(C[p_1]) \leq \mathcal{C}(C[p_2])$$

L'équivalence correspondante est notée : $\sqsupseteq\!\!\!\sqsubseteq$.

On peut d'ores et déjà énoncer quelques propriétés de $\sqsubseteq$:

Proposition 2.5.2.

1. $\forall p_1, p_2, \quad p_1 \sqsubseteq p_2 \Rightarrow (\mathcal{C}(p_1), \Delta(p_1)) \leq^2 (\mathcal{C}(p_2), \Delta(p_2))$

2. $\epsilon \sqsupseteq\!\!\!\sqsubseteq (0,0)$

3. $\forall (c_1,\delta_1),(c_2,\delta_2), \quad (c_1,\delta_1) \leq^2 (c_2,\delta_2) \Leftrightarrow (c_1,\delta_1) \sqsubseteq (c_2,\delta_2)$

4. $\forall (c_1,\delta_1),(c_2,\delta_2), \quad (c_1,\delta_1)(c_2,\delta_2) \sqsubseteq (\max(c_1, c_2+\delta_1), \delta_1+\delta_2)$

5. Si $p \xrightarrow{l} p'$ *alors* $p \sqsubseteq lp'$*. En particulier,* $p \xrightarrow{l} \epsilon$ *implique* $p \sqsubseteq l$*.*

où l'on a noté $\leq^2$ *le pré-ordre produit "composante par composante" sur* $\mathbb{N}\times\mathbb{Z}$ *:* $(c_1,\delta_1) \leq^2 (c_2,\delta_2)$ *ssi* $c_1 \leq c_2$ *et* $\delta_1 \leq \delta_2$*.*

Démonstration.

1. $\mathcal{C}(p_1) \leq \mathcal{C}(p_2)$ est évident. Pour montrer $\Delta(p_1) \leq \Delta(p_2)$, on considère le contexte $\_(M,0)$ pour M assez grand.
2. Soit $C[\_]$ un contexte, et $C[(0,0)] \xrightarrow{l_0} \epsilon$ un ordonnancement optimal pour $C[(0,0)]$. D'après la proposition 2.1.2, ce chemin s'écrit $l_0 = l_1(0,0)l_2$, avec :

$$\begin{cases} C[\_] \xrightarrow{l_1} E[\_] \\ E[(0,0)] \xrightarrow{(0,0)} E[\epsilon] \xrightarrow{l_2} \epsilon \end{cases}$$

Or dans ce cas, on a également :

$$C[\epsilon] \xrightarrow{l_1} E[\epsilon] \xrightarrow{l_2} \epsilon$$

D'où, $\mathcal{C}(C[\epsilon]) \leq \mathcal{C}(l_1 l_2) = \mathcal{C}(l_1(0,0)l_2) = \mathcal{C}(C[(0,0)])$. Inversement, un ordonnancement optimal de $C[\epsilon]$ s'écrit $l'_1 l'_2$:

$$\begin{cases} C[\_] \xrightarrow{l'_1} E[\_] \\ E[\epsilon] \xrightarrow{l'_2} \epsilon \end{cases}$$

d'où on déduit :

$$C[(0,0)] \xrightarrow{l'_1} E[(0,0)] \xrightarrow{(0,0)} E[\epsilon] \xrightarrow{l'_2} \epsilon$$

et finalement : $\mathcal{C}(C[\epsilon]) = \mathcal{C}(C[(0,0)])$.

3. le sens $\Leftarrow$ découle du 1. Réciproquement, soit $C[\_]$ un contexte. Un ordonnancement optimal pour $C[(c_2, \delta_2)]$ s'écrit $l_0 = l_1(c_2, \delta_2)l_2$, avec :

$$\begin{cases} C[\_] \xrightarrow{l_1} E[\_] \\ E[(c_2, \delta_2)] \xrightarrow{(c_2,\delta_2)} E[\epsilon] \xrightarrow{l_2} \epsilon \end{cases}$$

Or dans ce cas, on a également :

$$C[(c_1, \delta_1)] \xrightarrow{l_1} E[(c_1, \delta_1)] \xrightarrow{(c_1,\delta_1)} E[\epsilon] \xrightarrow{l_2} \epsilon$$

D'où :

$$\mathcal{C}(C[(c_1, \delta_1)]) \leq \mathcal{C}(l_1(c_1, \delta_1)l_2) \leq \mathcal{C}(l_1(c_2, \delta_2)l_2) = \mathcal{C}(C[(c_2, \delta_2)])$$

4. Soit $C[\_]$ un contexte et l_0 un ordonnancement optimal pour $C[(\max(c_1, c_2+\delta_1), \delta_1+\delta_2)]$. Comme précédemment, $l_0 = l_1(\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)l_2$ avec :

$$\begin{cases} C[\_] \xrightarrow{l_1} E[\_] \\ E[(\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)] \xrightarrow{(\max(c_1,c_2+\delta_1),\delta_1+\delta_2)} E[\epsilon] \xrightarrow{l_2} \epsilon \end{cases}$$

Par conséquent :

$$C[(c_1, \delta_1)(c_2, \delta_2)] \xrightarrow{l_1} E[(c_1, \delta_1)(c_2, \delta_2)] \xrightarrow{(c_1,\delta_1)(c_2,\delta_2)} E[\epsilon] \xrightarrow{l_2} \epsilon$$

D'où :

$$\begin{aligned} \mathcal{C}(C[(c_1, \delta_1)(c_2, \delta_2)]) &\leq \mathcal{C}(l_1(c_1, \delta_1)(c_2, \delta_2)l_2) \\ &= \mathcal{C}(l_1(\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)l_2) \\ &= \mathcal{C}(C[(\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)]). \end{aligned}$$

5. Soit $C[\_]$ un contexte et l_0 un ordonnancement optimal pour $C[lp']$. En examinant les transitions possible de lp', on se convainc de l'existence de listes l_1, l_2 et d'un contexte $E[\_]$ tels que :

$$\begin{cases} l_0 = l_1\, l_2 \\ C[lp'] \xrightarrow{l_1} E[p'] \xrightarrow{l_2} \epsilon \end{cases}$$

D'après la proposition 2.1.2, l_1 est une suite de transitions issue du contexte $C[\_]$ et de la liste l, on a également $C[p] \xrightarrow{l_1} E[p'] \xrightarrow{l_2} \epsilon$, d'où finalement $\mathcal{C}(p \parallel k) \leq \mathcal{C}(lp' \parallel k')$. □

2.5.2 Étude de la fusion, normalisation des listes

Dans le but de compresser les listes *sans perte d'information*, on cherche une condition nécessaire et suffisante pour que : $(c_1, \delta_1)(c_2, \delta_2) \sqsupseteq (\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)$ (sachant que l'on a déjà le sens $\sqsubseteq$).

Théorème 2.5.3. $(c_1, \delta_1)(c_2, \delta_2) \sqsupseteq (\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)$ *ssi :*

$$\begin{array}{r|l} & \delta_2 \leq 0 \;\; et\; c_1 \geq c_2 + \delta_1 \;\; (1) \\ ou & \delta_1 \geq 0 \;\; et\; c_1 \leq c_2 + \delta_1 \;\; (2) \end{array}$$

Démonstration. Montrons tout d'abord que la condition est nécessaire en considérant les contextes de la forme $(\_ \parallel (c_3, \delta_3))$ (avec $c_3 \geq \max(0, \delta_3)$).

Notons $(c_{12}, \delta_{12}) = (\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)$. En examinant les différents ordonnancements pour $(c_1, \delta_1)(c_2, \delta_2) \parallel (c_3, \delta_3)$ et $(c_{12}, \delta_{12}) \parallel (c_3, \delta_3)$, on obtient que la condition

$$\mathcal{C}(\,(c_1, \delta_1)(c_2, \delta_2) \parallel (c_3, \delta_3)\,) \geq \mathcal{C}(\,(c_{12}, \delta_{12}) \parallel (c_3, \delta_3)\,)$$

équivaut à :

$$\max(c_1, c_3+\delta_1, c_2+\delta_1+\delta_3) \geq \min(\max(c_1, c_2+\delta_1, c_3+\delta_1+\delta_2), \max(c_3, c_1+\delta_3, c_2+\delta_1+\delta_3))$$

Prenons en particulier c_3 et δ_3 tels que $c_1 = c_3 + \delta_1 = c_2 + \delta_1 + \delta_3$, autrement dit :

$$\begin{cases} c_3 = c_1 - \delta_1 \geq 0 \\ \delta_3 = c_3 - c_2 = c_1 - \delta_1 - c_2 \leq c_3 \end{cases}$$

L'équation précédente s'écrit alors :

$$c_1 \geq \min(\max(c_2 + \delta_1, c_1 + \delta_2), \max(c_1 - \delta_1, 2c_1 - \delta_1 - c_2)$$

ce qui équivaut à la condition cherchée.

Réciproquement la condition est suffisante. Soit en effet un contexte $C[\_]$ quelconque et un l_0 un chemin optimal pour $C[(c_1, \delta_1)(c_2, \delta_2)]$. D'après la proposition 2.1.2, l_0 peut se mettre sous la forme $l_0 = l_1(c_1, \delta_1)l_2(c_2, \delta_2)l_3$, avec :

$$\begin{cases} C[\_] \xrightarrow{l_1} E[\_] \\ E[(c_1, \delta_1)(c_2, \delta_2)] \xrightarrow{(c_1, \delta_1)} E[(c_2, \delta_2)] \\ E'[\_] \xrightarrow{l_2} E'[\_] \\ E'[(c_2, \delta_2)] \xrightarrow{(c_2, \delta_2)} E'[\epsilon] \xrightarrow{l_3} \epsilon \end{cases}$$

On dispose donc des deux réductions suivantes :

$$\begin{cases} C[(c_{12}, \delta_{12})] \xrightarrow{l_1} E[(c_{12}, \delta_{12})] \xrightarrow{(c_{12}, \delta_{12})} E[\epsilon] \xrightarrow{l_2} E'[\epsilon] \xrightarrow{l_3} \epsilon \\ C[(c_{12}, \delta_{12})] \xrightarrow{l_1} E[(c_{12}, \delta_{12})] \xrightarrow{l_2} E'[(c_{12}, \delta_{12})] \xrightarrow{(c_{12}, \delta_{12})} E'[\epsilon] \xrightarrow{l_3} \epsilon \end{cases}$$

Or, en notant $(c_3, \delta_3) = (\mathcal{C} \times \Delta)(l_2)$, la condition implique :

$$\max(c_1, c_3+\delta_1, c_2+\delta_1+\delta_3) \geq \min(\max(c_1, c_2+\delta_1, c_3+\delta_1+\delta_2), \max(c_3, c_1+\delta_3, c_2+\delta_1+\delta_3))$$

c'est-à-dire :

$$\mathcal{C}((c_1, \delta_1)l_2(c_2, \delta_2)) \geq \min(\mathcal{C}((c_{12}, \delta_{12})l_2),\ \mathcal{C}(l_2(c_{12}, \delta_{12})))$$

On obtient donc finalement :

$$\begin{aligned} \mathcal{C}(C[(c_{12}, \delta_{12})]) &\leq \min(\mathcal{C}(l_1(c_{12}, \delta_{12})l_2l_3),\ \mathcal{C}(l_1l_2(c_{12}, \delta_{12})l_3)) \\ &\leq \mathcal{C}(l_1(c_1, \delta_1)l_2(c_2, \delta_2)l_3) \\ &= \mathcal{C}(C[(c_1, \delta_1)(c_2, \delta_2)]) \end{aligned}$$

□

Définition 2.5.4. On notera désormais $(c_1, \delta_1)\mathcal{S}(c_2, \delta_2)$ cette propriété de *simplifiabilité*. La restriction de $\xrightarrow{fusion}$ aux couples simplifiables est notée $\xrightarrow{simpl}$.

Proposition 2.5.5. $\xrightarrow{simpl}$ *est confluente.*

Démonstration. Comme précédemment, la paire critique s'écrit :

$$\begin{tikzcd}
(c_1, \delta_1)(c_2, \delta_2)(c_3, \delta_3) \arrow[r] \arrow[d] & (\max(c_1, \delta_1 + c_2), \delta_1 + \delta_2)(c_3, \delta_3) \arrow[d, "?"] \\
(c_1, \delta_1)(\max(c_2, \delta_2 + c_3), \delta_2 + \delta_3) \arrow[r, "?"] & (\max(c_1, \delta_1 + c_2, \delta_1 + \delta_2 + c_3), \delta_1 + \delta_2 + \delta_3)
\end{tikzcd}$$

On voudrait donc montrer :

$$\begin{cases} (c_1, \delta_1)\mathcal{S}(c_2, \delta_2) & (H_{12}) \\ (c_2, \delta_2)\mathcal{S}(c_3, \delta_3) & (H_{23}) \end{cases} \Longrightarrow \begin{cases} (\max(c_1, \delta_1 + c_2), \delta_1 + \delta_2)\mathcal{S}(c_3, \delta_3) & (H_{(12)3}) \\ (c_1, \delta_1)\mathcal{S}(\max(c_2, \delta_2 + c_3), \delta_2 + \delta_3) & (H_{1(23)}) \end{cases}$$

Pour ce faire, raisonnons par cas selon le contenu de H_{12} et de H_{23} :

$H_{23} \diagdown H_{12}$	$\delta_2 \le 0$ $c_1 \ge c_2 + \delta_1$	$\delta_1 \ge 0$ $c_1 \le c_2 + \delta_1$
$\delta_3 \le 0$ $c_2 \ge c_3 + \delta_2$	$\begin{cases} \delta_3 \le 0 \\ \max(c_1, c_2 + \delta_1) \ge c_3 + \delta_1 + \delta_2 \end{cases}$ $\begin{cases} \delta_2 + \delta_3 \le 0 \\ c_1 \ge \max(c_2, c_3 + \delta_2) + \delta_1 \end{cases}$	$\begin{cases} \delta_3 \le 0 \\ \max(c_1, c_2 + \delta_1) \ge c_3 + \delta_1 + \delta_2 \end{cases}$ $\begin{cases} \delta_1 \ge 0 \\ c_1 \le \max(c_2, c_3 + \delta_2) + \delta_1 \end{cases}$
$\delta_2 \ge 0$ $c_2 \le c_3 + \delta_2$	?	$\begin{cases} \delta_1 + \delta_2 \ge 0 \\ \max(c_1, c_2 + \delta_1) \le c_3 + \delta_1 + \delta_2 \end{cases}$ $\begin{cases} \delta_1 \ge 0 \\ c_1 \le \max(c_2, c_3 + \delta_2) + \delta_1 \end{cases}$

La case marquée ? semble poser problème, car l'on ne parvient à montrer ni $(H_{(12)3})$ ni $(H_{1(23)})$. En réalité, les conditions vérifiées dans cette case s'écrivent :

$$\begin{cases} \delta_2 = 0 \\ c_2 \le c_3 \\ c_1 \ge c_2 + \delta_1 \end{cases}$$

ce qui implique :

$$(\max(c_1, \delta_1 + c_2), \delta_1 + \delta_2)(c_3, \delta_3) = (c_1, \delta_1)(\max(c_2, \delta_2 + c_3), \delta_2 + \delta_3)$$

Autrement dit, dans ce cas le diagramme converge dès la première étape, et il n'y a rien à vérifier (!). □

Définition 2.5.6. Une liste de blocs en forme normale pour $\xrightarrow{simpl}$ et distincte de $(0, 0)$ sera dite *normalisée*.

Remarque. La condition "différent de $(0, 0)$" permet d'éliminer la redondance entre $(0, 0)$ et ϵ.

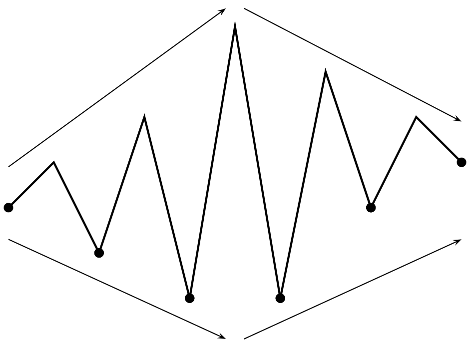
Dans une liste normalisée, deux blocs consécutifs $(c_1, \delta_1)(c_2, \delta_2)$ vérifient :

$$\begin{cases} \delta_2 > 0 \text{ ou } c_1 < c_2 + \delta_1 \\ \delta_1 < 0 \text{ ou } c_1 > c_2 + \delta_1 \end{cases}$$

c'est-à-dire :

$$\begin{array}{l|l} & \delta_1 < 0 \text{ et } \delta_2 > 0 \\ \text{ou} & \delta_1 < 0 \text{ et } c_1 < c_2 + \delta_1 \\ \text{ou} & \delta_2 > 0 \text{ et } c_1 > c_2 + \delta_1 \end{array}$$

Ces inégalités peuvent s'interpréter comme des contraintes sur le graphe de variation des listes normalisées (voir figure 2.3).



$(1, -1)$ $(3, -1)$ $(6, 0)$ $(5, 2)$ $(2, 1)$

FIG. 2.3 – Exemple de liste normalisée

Une autre propriété remarquable des listes normalisées est d'être *triées selon* $\preceq$ (et même strictement croissantes). En effet :

Proposition 2.5.7. *$(\mathcal{S}) \supseteq (\succeq)$, ce qui équivaut à $(\not\mathcal{S}) \subseteq (\prec)$ puisque $\preceq$ est un ordre total.*

Démonstration. Supposons $(c_1, \delta_1) \succeq (c_2, \delta_2)$ et montrons :

$$\begin{array}{l|l} & \delta_2 \leq 0 \text{ et } c_1 \geq c_2 + \delta_1 \quad (1) \\ \text{ou} & \delta_1 \geq 0 \text{ et } c_1 \leq c_2 + \delta_1 \quad (2) \end{array}$$

3 cas sont possibles :

1. si $\delta_2 \leq 0$ et $\delta_1 \geq 0$, alors suivant si $c_1 \geq c_2 + \delta_1$ ou si $c_1 \leq c_2 + \delta_1$, (1) ou (2) est vrai.
2. si $\delta_2 < 0$, $\delta_1 < 0$ et $c_2 \leq c_1$, alors (1) est vrai
3. si $\delta_2 > 0$, $\delta_1 > 0$ et $\delta_2 - c_2 \leq \delta_1 - c_1$, alors (2) est vrai □

Discussion Ce résultat s'avère particulièrement intéressant en relation avec le théorème 2.4.8 sur l'ordonnancement des listes triées. En effet, on dispose maintenant d'un algorithme pour calculer $\mathcal{C}(l_1 \parallel l_2 \parallel \ldots \parallel l_n)$ lorsque les l_i sont des listes quelconques (il suffit de les normaliser, puis d'appliquer le théorème 2.4.8).

Pour pouvoir calculer $\mathcal{C}(p_1 \parallel p_2 \parallel \ldots \parallel p_n)$ lorsque les p_i ne sont pas forcément des listes, l'idéal serait de trouver un moyen de traduire des processus en listes (normalisées) pour se ramener au cas précédent.

En supposant $l_1, l_2, \ldots, l_n$ normalisées, un candidat idéal pour traduire $l_1 \parallel l_2 \parallel \ldots \parallel l_n$ serait $merge_\preceq(l_1, l_2, \ldots, l_n)$. En effet, on a déjà montré l'égalité des coûts, et de plus la proposition 2.5.2-5 implique :

$$l_1 \parallel l_2 \parallel \ldots \parallel l_n \sqsupseteq merge_\preceq(l_1, l_2, \ldots, l_n)$$

Avant de montrer l'équivalence, nous allons chercher à simplifier la définition de $\sqsubseteq$.

2.5.3 Simplification de $\sqsubseteq$

Lemme 2.5.8. *Supposons* $E_0[\_] \xrightarrow{x_1} E_1[\_] \xrightarrow{x_2} \dots \xrightarrow{x_n} E_n[\_]$ *et soit* $l = x_1\, x_2 \dots x_n$. *Alors :*

$$\forall p, l', \quad p \| l \xrightarrow{l'} \epsilon \Rightarrow E_0[p] \xrightarrow{l'} E_n[\epsilon]$$

Démonstration. Comme $p\|l \xrightarrow{l'} \epsilon$, il existe $l_1, \dots, l_{n+1}$, $p_0, \dots, p_{n+1}$ tels que :

$$\begin{cases} p = p_0 \xrightarrow{l_1} p_1 \xrightarrow{l_2} \dots \xrightarrow{l_{n+1}} p_{n+1} = \epsilon \\ l' = l_1\, x_1\, l_2\, x_2\, \dots\, l_n\, x_n\, l_{n+1} \end{cases}$$

On a alors :

$$\begin{array}{rcll} E_0[p] = E_0[p_0] & \xrightarrow{l_1} & E_0[p_1] \xrightarrow{x_1} E_1[p_1] \\ & \xrightarrow{l_2} & E_1[p_2] \xrightarrow{x_2} E_2[p_2] \\ & \dots & \\ & \xrightarrow{l_n} & E_{n-1}[p_n] \xrightarrow{x_n} E_n[p_n] \\ & \xrightarrow{l_{n+1}} & E_n[p_{n+1}] = E_n[\epsilon] \end{array}$$

□

Lemme 2.5.9. *Pour tout contexte* $C[\_]$, *pour tout processus* p_0, *il existe des listes* l_0, l_1, l_2 *telles que :*

$$\begin{cases} \mathcal{C}(C[p_0]) = \mathcal{C}(l_0(p_0 \parallel l_1)l_2) \\ \forall p, \quad \mathcal{C}(C[p]) \leq \mathcal{C}(l_0(p \parallel l_1)l_2) \end{cases}$$

Démonstration. Soit l_{opt} un ordonnancement optimal pour $C[p_0]$. En appliquant les propositions 2.1.1 et 2.1.2 de manière répétée, on peut mettre l_{opt} sous la forme $l_{opt} = l_0\, l'_1\, l''_1\, l'_2\, l''_2 \dots l'_{m-1}\, l''_{m-1}\, l'_m\, l_2$, avec [3] :

$$\begin{array}{lcllcl} C[p_0] & \xrightarrow{l_0} & E_1[p_0] & & & \\ & \xrightarrow{l'_1} & E_1[p_1] & & \xrightarrow{l''_1} & E_2[p_1] \\ & \xrightarrow{l'_2} & E_2[p_2] & & \xrightarrow{l''_2} & E_3[p_2] \\ & \dots & & & & \\ & \xrightarrow{l'_{m-1}} & E_{m-1}[p_{m-1}] & & \xrightarrow{l''_{m-1}} & E_m[p_2] \\ & \xrightarrow{l'_m} & E_m[\epsilon] & & & \\ & \xrightarrow{l_2} & \epsilon & & & \end{array}$$

Soit alors $l_1 = l''_1\, l''_2 \dots l''_{m-1}$. Comme $l_0(p_0 \parallel l_1)l_2 \xrightarrow{l_{opt}} \epsilon$, on sait déjà que $\mathcal{C}(C[p_0]) \geq \mathcal{C}(l_0(p_0 \parallel l_1)l_2)$.

Montrons maintenant la deuxième partie de l'affirmation. Soit donc p quelconque. Un ordonnancement – en particulier optimal – pour $l_0(p \parallel l_1)l_2$ est nécessairement de la forme $l_0\, l\, l_2$ avec $p \parallel l_1 \xrightarrow{l} \epsilon$. Étant le lemme précédent, ceci implique $E_1[p] \xrightarrow{l} E_m[\epsilon]$, d'où $C[p] \xrightarrow{l_0\, l\, l_2} \epsilon$. □

[3]Les réductions de la forme $C[p] \xrightarrow{l} C'[p]$ signifiant implicitement $C[\_] \xrightarrow{l} C'[\_]$.

Lemme 2.5.10. *Soit p un programme. Pour M assez grand,*

$$\mathcal{C}(p \parallel (M,M)) = \mathcal{C}(p\,(M,M))$$

Démonstration. Soit $l = l_1(M,M)l_2$ un ordonnancement optimal pour $p \parallel (M,M)$. Si $M \geq \mathcal{C}(l_2)$, alors :

$$\begin{aligned}
\mathcal{C}(l_1 l_2(M,M)) &= \max(\mathcal{C}(l_1), \mathcal{C}(l_2) + \Delta(l_1), M + \Delta(l_1) + \Delta(l_2)) \\
&\leq \max(\mathcal{C}(l_1), M + \Delta(l_1), \mathcal{C}(l_2) + \Delta(l_1) + M) \\
&= \mathcal{C}(l_1(M,M)l_2)
\end{aligned}$$

□

On peut maintenant énoncer une définition plus simple de $\sqsubseteq$:

Théorème 2.5.11. *Pour tous p_1, p_2,*

$$p_1 \sqsubseteq p_2 \text{ ssi } \forall l, \mathcal{C}(p_1 \parallel l) \leq \mathcal{C}(p_2 \parallel l)$$

Démonstration. Montrons l'implication $\Leftarrow$. Si l'on choisit $l = (M,M)$ avec M assez grand, d'après le lemme précédent :

$$\mathcal{C}(p_i \parallel (M,M)) = \mathcal{C}(p_i(M,M)) = \Delta(p_i) + M \qquad (i = 1,2)$$

D'où on déduit, $\Delta(p_1) \leq \Delta(p_2)$.

Soit alors $C[\_]$ un contexte quelconque. On choisit l_0, l_1, l_2 adaptées à $C[\_]$ et à p_2 au sens du lemme précédent. Compte tenu de ce qui précède, on a :

$$\begin{aligned}
\mathcal{C}(C[p_1]) &\leq \mathcal{C}(l_0(p_1 \parallel l_1)l_2) \\
&= \max(\mathcal{C}(l_0), \Delta(l_0) + \mathcal{C}(p_1 \parallel l_1), \Delta(l_0) + \Delta(l_1) + \Delta(p_1) + \mathcal{C}(l_2)) \\
&\leq \max(\mathcal{C}(l_0), \Delta(l_0) + \mathcal{C}(p_2 \parallel l_1), \Delta(l_0) + \Delta(l_1) + \Delta(p_2) + \mathcal{C}(l_2)) \\
&= \mathcal{C}(l_0(p_2 \parallel l_1)l_2) \\
&= \mathcal{C}(C[p_2])
\end{aligned}$$

□

Remarque. Compte-tenu des résultats sur la normalisation des listes, on peut énoncer le même théorème mais portant uniquement sur des listes *normalisées.*

2.5.4 Traduction de l'opérateur $\parallel$

Proposition 2.5.12. *Si les $l_i = y_i l'_i$ $(i = 1, \ldots, n)$ sont des listes* triées, *et si $y_1 \preceq y_i$ pour tout i alors*

$$l_1 \parallel \ldots \parallel l_n \sqsupseteq y_1(l'_1 \parallel l_2 \parallel \ldots \parallel l_n)$$

Démonstration. Étant donné que $(l_1 \parallel \ldots \parallel l_n) \xrightarrow{y_1} (l'_1 \parallel l_2 \parallel \ldots \parallel l_n)$, la proposition 2.5.2-5 nous donne déjà : $l_1 \parallel \ldots \parallel l_n \sqsubseteq y_1(l'_1 \parallel l_2 \parallel \ldots \parallel l_n)$

Soit maintenant $l_0 = y_0 l'_0$ triée quelconque. Si $y_1 \preceq y_0$, alors :

$$\begin{aligned}
\mathcal{C}(l_1 \parallel \ldots \parallel l_n \parallel l_0) &= \mathcal{C}(y_1(l'_1 \parallel \ldots \parallel l_n \parallel l_0)) && \text{(lemme 2.4.7)} \\
&\geq \mathcal{C}((y_1(l'_1 \parallel \ldots \parallel l_n)) \parallel l_0) && \text{(proposition 2.5.2-5)}
\end{aligned}$$

Si $y_0 \preceq y_1$, alors :

$$\begin{aligned}
\mathcal{C}(l_1 \parallel \ldots \parallel l_n \parallel l_0) &= \mathcal{C}(y_0 y_1(l'_1 \parallel \ldots \parallel l_n \parallel l'_0)) && \text{(lemme 2.4.7)} \\
&\geq \mathcal{C}((y_1(l'_1 \parallel \ldots \parallel l_n)) \parallel l_0) && \text{(proposition 2.5.2-5)}
\end{aligned}$$

□

En considérant une définition inductive de $merge_{\preceq}(l_1, l_2, \ldots, l_n)$, on peut maintenant établir :

Théorème 2.5.13. *Si $l_1, l_2, \ldots, l_n$ sont triées,*

$$l_1 \parallel l_2 \parallel \ldots \parallel l_n \sqsupseteq\sqsubseteq merge_{\preceq}(l_1, l_2, \ldots, l_n)$$

Corollaire 2.5.14 (Correction de la traduction 1).

$$\forall p, \quad [\![p]\!]^1 \sqsupseteq p$$

Démonstration. Étant donné l'étude de la fusion et le théorème précédent, cette propriété se montre immédiatement par induction sur p. □

2.5.5 Une traduction exacte vers les listes normalisées

On possède désormais tous les éléments pour énoncer une traduction exacte des processus vers les listes normalisées :

$$\begin{aligned}
[\![\epsilon]\!]^2 &\stackrel{déf}{=} \epsilon \\
[\![(c,\delta)]\!]^2 &\stackrel{déf}{=} (c,\delta) \\
[\![p_1 p_2]\!]^2 &\stackrel{déf}{=} simpl^*([\![p_1]\!]^2 [\![p_2]\!]^2) \\
[\![l_1 \parallel l_2]\!]^2 &\stackrel{déf}{=} simpl^*(merge_{\preceq}([\![l_1]\!]^2, [\![l_2]\!]^2))
\end{aligned}$$

Théorème 2.5.15 (Correction et complétude de la traduction 2).

$$\forall p, \quad [\![p]\!]^2 \sqsupseteq\sqsubseteq p$$

Démonstration. par induction sur p, grâce aux théorèmes 2.5.3, 2.5.13 et 2.5.7 □

A l'aide de cette traduction, le problème du calcul du coût se résout maintenant de manière immédiate :

Corollaire 2.5.16 (Algorithme de calcul du coût).

$$\forall p, \quad \mathcal{C}(p) = \mathcal{C}([\![p]\!]^2)$$

2.5.6 Retour aux variations

Soit $\mathcal{P}_x$ et $\mathcal{P}_y$ l'ensemble des programmes, respectivement à variations et à blocs. Il est clair que l'on peut également définir $\sqsubseteq$ sur $\mathcal{P}_x$. La correspondance entre variations et blocs apparaît alors comme une équivalence d'espaces pré-ordonnés :

Théorème 2.5.17.

$$\begin{array}{ll} \forall p_x, p'_x, & p_x \sqsubseteq p'_x \Rightarrow [\![p_x]\!]^0 \sqsubseteq [\![p'_x]\!]^0 \\ \forall p_y, p'_y, & p_y \sqsubseteq p'_y \Rightarrow [\![p_y]\!]^{0'} \sqsubseteq [\![p'_y]\!]^{0'} \\ \forall p_x, & p_x \sqsupseteq [\![[\![p_x]\!]^0]\!]^{0'} \\ \forall p_y, & p_y \sqsupseteq [\![[\![p_y]\!]^{0'}]\!]^0 \end{array}$$

ce que l'on note :

$$(\mathcal{P}_x, \sqsubseteq) \underset{0}{\overset{0'}{\approx}} (\mathcal{P}_y, \sqsubseteq)$$

Les résultats sur les blocs se transposent donc immédiatement aux variations.

En fait, le théorème de corrections de la traductions $[\![\_]\!]^2$ s'interprétent également en termes d'équivalence d'espaces pré-ordonnés :

Théorème 2.5.18. *Soit $\mathcal{L}$ l'espace des listes normalisés, muni du pré-ordre $\sqsubseteq$ (dont on sait depuis le théorème 2.5.11 qu'il peut se définir uniquement par les listes). Soit ι l'injection des listes dans les processus. On a l'équivalence suivante :*

$$(\mathcal{P}_y, \sqsubseteq) \underset{2}{\overset{\iota}{\approx}} (\mathcal{L}, \sqsubseteq)$$

La composition des traduction $[\![\_]\!]^0$ et $[\![\_]\!]^1$ (resp. $[\![\_]\!]^2$) sera notée $[\![\_]\!]^{01}$ (resp. $[\![\_]\!]^{02}$).

2.5.7 Commentaires

Interprétation de la traduction La traduction d'un processus en liste normalisée consiste d'une certaine manière à fixer un ordonnancement "optimal au sens fort". Un ordonnancement optimal quelconque ne convient pas toujours dans la mesure où l'optimalité est une propriété globale qui ne dit rien sur les étapes intermédiaires.

Cette opération est rendue possible par des propriétés remarquables sur les contextes, que l'on peut attribuer intuitivement à l'absence d'interaction entre les processus (celles-ci ont été "externalisées" via les transitions étiquetées).

Choix du critère de simplification Un autre candidat pour le critère de simplification aurait pu être simplement $\mathcal{S}' \overset{déf}{=} (\succ)$, ce qui revient à prendre le critère le plus restreint possible et non le plus large possible.

On montre en effet que la règle $\xrightarrow{simpl'}$ ainsi obtenue est encore confluente (proposition B.0.2 en annexe). Comme $(\mathcal{S}) \supseteq (\mathcal{S}')$ et comme les nouvelles listes normalisées sont (évidemment) triées, la traduction correspondante $[\![\_]\!]^3$ est encore correcte et complète.

Toutefois, cette traduction est moins efficace – en espace *et* en temps – que $[\![\_]\!]^2$ dans la mesure où les listes sont moins compressées. On utilisera donc la traduction $[\![\_]\!]^2$ meilleure à tout point de vue.

Complexité de l'algorithme Les calculs de $simpl^*(merge_{\preceq}(l_1, l_2))$ et de $simpl^*(l_1\, l_2)$ s'effectuent tout deux en temps $O(n_1 + n_2)$ où n_i est la taille de la liste l_i. Si m est la profondeur de l'arbre syntaxique (binaire) de p et n sa taille, la traduction de p s'effectue donc en temps $O(nm)$.

En pratique, les opérations de fusion diminuent très notablement la taille des listes à manipuler (voir l'étude expérimentale en annexe à la section A.2) de sorte que l'algorithme de traduction a un comportement quasi-linéaire.

Chapitre 3

Application à l'ordonnancement de tâches

Nous allons maintenant décrire une architecture permettant à un ordonnanceur de tâches d'éviter les interblocages sur la mémoire.

Comme prévu, la solution comprend deux parties :

- un module d'analyse statique de programmes, pour extraire les informations de consommation mémoire.
- un module dynamique, dont le rôle est de prévenir les interblocages à l'aide des informations obtenues lors de la phase statique.

La communication entre le premier et le second module se fait par des annotations ajoutées au cœur du programme.

3.1 Analyse statique des consommations

L'analyse statique repose tout d'abord sur une analyse de valeurs, qui a pour but de majorer les quantités de mémoire allouées (ou de minorer les quantités désallouées). Cette analyse dépend bien sûr du langage considéré, et peut être réalisée de diverses manières :

1. par typage, si le langage dispose d'un système de types dédié au contrôle de ressource ;
2. par interprétation abstraite, dans les autres cas.

Les langages possédant un système de types pour le contrôle des ressources restent rares ; on peut citer par exemple les travaux de [CW00] sur un système de types dépendants permettant de borner le temps d'exécution des fonctions.

L'interprétation abstraite, introduite par [CC77], est un cadre général pour l'analyse de programmes. On pourra se reporter à [CC92] ou [Cou99] pour une introduction à la théorie et des exemples d'analyseur.

Supposons désormais mis en place un analyseur statique permettant de majorer les quantités allouées. Sous réserve que les structures du langage source soit traduisibles, on peut alors approximer chaque programme par un élement du langage $\mathcal{P}_x$ introduit au chapitre 2. Dans l'état actuel du formalisme, ce langage peut certes sembler insuffisant (absence de boucles, de branchements...etc) ; on verra au chapitre 5 des pistes pour combler ces lacunes.

Dans le cadre de l'interprétation abstraite, l'approximation d'une sémantique $(\mathcal{S}_0, \preceq_0)$ par une sémantique $(\mathcal{S}_1, \preceq_1)$ est formalisé par une paire de fonctions $\alpha : \mathcal{S}_0 \to \mathcal{S}_1$ et $\gamma : \mathcal{S}_1 \to \mathcal{S}_0$ telles que :

$$\forall x \in \mathcal{S}_0, y \in \mathcal{S}_1, \quad \alpha(x) \preceq_1 y \Longleftrightarrow x \preceq_0 \gamma(y)$$

Une telle paire est appelée *connexion de Galois* et notée :

$$(\mathcal{S}_0, \preceq_0) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{S}_1, \preceq_1)$$

Soit $(\mathcal{S}, \preceq)$ une sémantique du langage de programmation analysé, et supposons construite une connexion de Galois (α, γ) entre $(\mathcal{S}, \preceq)$ et $(\mathcal{P}_x, \sqsubseteq)$. D'après le chapitre 2, on dispose alors de la chaîne d'approximations :

$$(\mathcal{S}, \preceq) \leftrightarrows (\mathcal{P}_x, \sqsubseteq) \underset{0}{\overset{0'}{\approx}} (\mathcal{P}_y, \sqsubseteq) \underset{2}{\overset{\iota}{\approx}} (\mathcal{L}, \sqsubseteq)$$

Autrement dit, les résultats précédents nous permettent de représenter le comportement global du programme analysé par une liste normalisée.

Dans l'optique d'une stratégie d'évitement de deadlock, une seule liste "globale" par programme ne suffit pas. En effet, pour détecter les zones dangereuses au cours de l'exécution il nous faut connaître à chaque instant une prévision sur les consommations mémoire à venir. Par conséquent, l'analyse doit associer une liste *à chaque point du programme*.

En pratique, ces listes sont calculées en une passe par un parcours du graphe de flot *vers l'arrière*, et en normalisant les listes au fur et à mesure. L'état prévisionnel du programme étant constant entre deux allocations, les listes ne sont ajoutées au programme – en guise d'annotations – qu'à l'endroit des allocations.

3.2 Phase dynamique : une librairie pour détecter les interblocages

La phase dynamique a pour but de garantir à chaque instant la sûreté de l'état courant, c'est-à-dire l'existence d'au moins une solution pour terminer toutes les tâches. Pour cela, il n'est pas nécessaire de réécrire l'ordonnanceur de tâches du système ; en fait, on peut conserver l'ordonnanceur de tâches existant et implémenter le module dynamique sous la forme d'une librairie de fonctions.

Utilisation de la librairie Avant chaque modification du système (allocation, désallocation, démarrage d'un processus, fin d'un processus...), un processus se doit d'appeler une fonction particulière de la librairie. L'appel de fonction donne en paramètres les informations prévisionnelles issues de l'analyse statique et concernant la suite du programme. Si la requête est considérée comme non-sûre, la fonction entre dans une boucle d'attente, ce qui a pour effet de bloquer le processus correspondant. Dans le cas contraire, la fonction met à jour l'état interne de la libraire puis rend la main au programme.

Fonctionnement Afin de détecter les états dangereux, la librairie connaît la quantité de mémoire restante dans le système et tient à jour une liste des consommations prévisionnelles des processus en cours. A chaque requête, elle simule la transition en calculant la nouvelle liste de consommations ainsi que la future quantité de mémoire disponible. Il suffit ensuite d'appliquer l'algorithme d'ordonnancement des listes pour vérifier que l'état correspondant est sûr. Si c'est le cas, la transition a lieu immédiatement ; dans le cas contraire, le processus demandeur est mis en attente et la liste des prévisions est restaurée dans son état initial.

Pour éviter les calculs inutiles, la boucle d'attente d'un processus est "semi-active" : les processus en attente ne sont réveillés – pour tenter à nouveau leur chance – que lorsque l'état du serveur a changé. Dans la plupart des langages, cette contrainte peut être implémentée efficacement par une variable de type "condition" et par les deux commandes "wait" et "broadcast".

3.3 Implémentation dans le cas séquentiel

Nous présentons maintenant un prototype d'ordonnanceur implémenté en Objective Caml [INR]. Par souci de simplicité, les processus sont de simples suites de variations et leur nombre est constant. Autrement dit, à chaque instant, l'ensemble des tâches est décrit par un terme de la forme $l_1\|l_2\|\ldots\|l_N$.

En entrée, le prototype prend la description des processus et la mémoire totale disponible. Il analyse alors leurs consommations respectives, puis simule leur exécution concurrente en suivant l'algorithme décrit au paragraphe précédent.

Étant donné les hypothèses de simplification du problème, la librairie du "serveur de mémoire" n'a que trois fonctions à implémenter :

- la commande $start(l)$ informe le serveur du démarrage d'un processus dont la liste prévisonnelle est l.
- la commande $alloc(x, l)$ demande à effectuer une allocation de x unités de mémoire (x entier relatif), sachant que la liste prévisionnelle *après allocation* sera l.
- la commande *end* informe le serveur de la fin du processus en cours.

Les arguments notés l, absents dans le programme initial, sont les résultats de l'analyse statique, que l'on peut résumer ainsi :

$$\begin{array}{lcl}
\text{start} & & \text{start}(simpl^*(\llbracket x_1\, x_2\, \ldots\, x_n \rrbracket^0)) \\
\text{alloc}(x_1) & & \text{alloc}(x_1,\, simpl^*(\llbracket x_2\, x_3\, \ldots\, x_n \rrbracket^0)) \\
\text{alloc}(x_2) & & \text{alloc}(x_2,\, simpl^*(\llbracket x_3\, x_4\, \ldots\, x_n \rrbracket^0)) \\
\ldots & \Longrightarrow & \ldots \\
\text{alloc}(x_{n-1}) & & \text{alloc}(x_{n-1},\, \llbracket x_n \rrbracket^0) \\
\text{alloc}(x_n) & & \text{alloc}(x_n,\, \epsilon) \\
\text{end} & & \text{end}
\end{array}$$

L'intérêt de la propriété de confluence de $simpl^*$ est de permettre un calcul en une passe, par la formule :

$$simpl^*(\llbracket x_k\, \ldots\, x_n \rrbracket^0) = simpl^*(\llbracket x_k \rrbracket^0\, simpl^*(\llbracket x_{k+1}\, \ldots\, x_n \rrbracket^0))$$

Théorème 3.3.1 (Correction de l'architecture). *Soit $l_1, \ldots, l_n$ les listes prévisionnelles initiales des processus et M la mémoire disponible du système. Si $\mathcal{C}(l_1\|\ldots\|l_n) \leq M$ alors le système parvient à mener toutes les tâches à leur terme.*

Démonstration. Par hypothèse, il existe un ordonnancement l de $l_1 \| \ldots \| l_n$ dont le coût est inférieur à M. Si $l = \epsilon$, c'est que toutes les tâches sont terminées. Sinon, $l = x\,l'$ et il existe une tâche l_i telle que $l_i \xrightarrow{x} l_i'$. Comme $\mathcal{C}(l) \leq M$, on a : $\mathcal{C}(l') \leq M - x$ et l'état $l_1 \| \ldots \| l_i' \| \ldots \| l_n$ (dont l' est un ordonnancement) est donc admissible. On conclue alors par récurrence. □

Résultats On trouvera un exemple d'exécution du prototype en annexe à la section A.3.1.

Complexité de l'algorithme Dans le cas purement séquentiel, chaque étape consiste à recalculer $\mathcal{C}(merge_{\preceq}(l_1, \ldots, l_N)$, où les l_i sont les listes de consommations prévisionnelles des tâches en cours.

Comme une seule des listes change à chaque étape, on peut diminuer le temps de calcul en gardant les résultats intermédiaires en mémoire. Notre implémentation utilise dans ce but des arbre binaires équilibrés "rouge-noir" (portant un bloc par nœud). Si m est la taille maximale des listes, la complexité de traitement d'une requête passe alors de $O(mN \log(N))$ à $O(m \log(mN))$ ce qui est intéressant car en pratique m dépasse rarement la dizaine du fait de la normalisation des listes.

Remarque. On pourrait aussi utiliser des arbres binaires équilibrés dont les informations aux nœuds sont des listes, plus précisément les résultats intermédiaires du calcul récursif :

$$merge_{\preceq}(l_1, \ldots, l_N) = merge_{\preceq}(merge_{\preceq}(l_1, \ldots, l_{\ulcorner N/2 \urcorner}), merge_{\preceq}(l_{\ulcorner N/2 \urcorner + 1}, \ldots, l_N) = \ldots$$

Dans ce cas, on montre que la complexité de l'algorithme est en $O(mN)$.

Chapitre 4

Le treillis des listes

Le résultat essentiel du chapitre 2, à savoir la traduction exacte des processus vers les listes (et même vers les listes normalisées), nous permet de restreindre l'étude de $\sqsubseteq$ à l'ensemble des listes. Nous allons maintenant en étudier la structure de manière plus approfondie.

D'une part, on aimerait savoir si les listes normalisées sont la plus petite structure de données pour représenter les classes d'équivalence de $\sqsupseteq\!\!\!\!\sqsubseteq$, autrement dit : est-ce que $l_1 \sqsupseteq\!\!\!\!\sqsubseteq l_2 \Rightarrow l_1 = l_2$ lorsque l_1 et l_2 sont normalisées ?

D'autre part, si l'on veut étendre un jour le formalisme à des programmes avec structures de contrôle, il serait très utile de posséder un opérateur de borne supérieur $\sqcup$.

4.1 Point de vue "absolu"

Jusqu'à présent les objets étudiés étaient des successions de transitions. Néanmoins la notion d'état était présente implicitement – ne serait-ce que dans la définition du coût :

$$\mathcal{C}(x_1 x_2 \ldots x_n) \stackrel{déf}{=} \max_{0 \leq i \leq n} \left(\sum_{1 \leq j < i} x_j \right) = \text{"max sur tous les états rencontrés"}$$

Remarque. La raison pour laquelle nous avons privilégié la présentation par transitions était double :

- d'une part, la formulation du problème initiale était "relative", car issue d'un langage de programmation.
- d'autre part, les ordonnancements étaient plus faciles à manipuler comme des entrelacements de variations que comme des chemins dans un graphe.

Afin d'étudier la relation $\sqsubseteq$ sur les listes, il nous faut maintenant aborder le point de vue "absolu".

4.1.1 Définitions

Définition 4.1.1 (État et vecteur d'états).

- Un *état* est un entier $a \in \mathbb{Z}$.
- On appelle *vecteur d'états* un élément $v = [a_0\, a_1\, \ldots\, a_n]$ de $\mathbb{Z}^{n+1}$. Le *coût* de v est par définition :

$$\mathcal{C}(v) = \max_{0 \leq i \leq n} a_i$$

Dans le but de définir $\mathcal{C}(v_1\|v_2)$, on introduit la notion de matrice somme et d'ordonnancement :

Définition 4.1.2 (Matrice d'états et ordonnancement).

- Si v_1 et v_2 sont deux vecteurs de taille $m+1$ et $n+1$, on note $v_1 \oplus v_2$ la matrice

$$A = [v_{1i} + v_{2j}] \in \mathbb{Z}^{\{0...m\}\times\{0...n\}}$$

- Soit $A = [a_{ij}] \in \mathbb{Z}^{\{0...m\}\times\{0...n\}}$. On appelle ordonnancement (ou chemin) de A toute fonction $\gamma : \{0\ldots m+n\} \to \{0\ldots m\} \times \{0\ldots n\}$ tel que :

$$\forall t < m+n,\ \gamma(t+1) - \gamma(t) \in \{(1,0),(0,1)\}$$

  Cette condition implique en particulier $\gamma(0) = (0,0)$ et $\gamma(T) = (m,n)$; on la note en abrégé $\gamma : (0,0) \rightsquigarrow (m,n)$.
  Les deux composantes de γ seront notées γ_1 et γ_2 (de sorte que $\gamma = \gamma_1 \times \gamma_2$).
  Remarque. Un raisonnement classique de dénombrement montre qu'il existe $\binom{m+n}{m} = \frac{(m+n)!}{m!\,n!}$ chemins possibles.
- Le vecteur $[a_{\gamma(0)} \ \ldots \ a_{\gamma(m+n)}] \in \mathbb{Z}^{\{0...m+n\}}$ est noté A_γ.
- Le *coût* de A est par définition :

$$\mathcal{C}(A) \stackrel{déf}{=} \min_\gamma \mathcal{C}(A_\gamma)$$

- Finalement on pose :

$$\mathcal{C}(v_1\|v_2) \stackrel{déf}{=} \mathcal{C}(v_1 \oplus v_2)$$

La définition du pré-ordre observationnel se transpose alors naturellement aux vecteurs d'états :

Définition 4.1.3 (Pré-ordre entre vecteurs).

$$v_1 \sqsubseteq v_2 \text{ ssi } \forall w,\ \mathcal{C}(v_1\|w) \leq \mathcal{C}(v_2\|w)$$

4.1.2 Correspondance entre listes et vecteurs

On définit une bijection ϕ entre les listes de variations et les vecteurs commençant par 0 :

$$\begin{aligned}\phi(x_1 \ldots x_n) &\stackrel{déf}{=} \left[\sum_{1\leq j\leq i} x_j\right]_{0\leq i\leq n}\\ \phi^{-1}([a_0\, a_1\ \ldots\ a_n]) &= (a_i - a_{i-1})_{1\leq i\leq n}\end{aligned}$$

Notons $lth(\_)$ la fonction taille pour les listes ou les vecteurs : $lth(x_1 \ldots x_n) = n$ et $lth([a_0 \ldots a_n]) = n+1$.

Etant donné les définitions précédentes, on montre sans mal le théorème suivant :

Théorème 4.1.4.

$$\forall l, \quad \begin{cases}\mathcal{C}(l) = \mathcal{C}(\phi(l))\\ \Delta(l) = \phi(l)_{lth(l)}\end{cases}$$

$$\forall l_1, l_2, \quad l_1 \sqsubseteq l_2 \Leftrightarrow \phi(l_1) \sqsubseteq \phi(l_2)$$

Remarque. Du point de vue des vecteurs, la règle de fusion des blocs s'interprète très simplement comme une suppression d'états. La règle :

$$(c_1, \delta_1)(c_2, \delta_2) \xrightarrow{fusion} (\max(c_1, c_2 + \delta_1), \delta_1 + \delta_2)$$

se transpose en effet :

$$[c_1\, \delta_1\, (\delta_1 + c_2)\, (\delta_1 + \delta_2)] \xrightarrow{fusion} [\max(c_1, c_2 + \delta_1)\, (\delta_1 + \delta_2)]$$

Conventions Si l est une liste de variations (ou de blocs) et a un entier, on note $a + l$ le vecteur $a + \phi(l)$ (ou $a + \phi(\llbracket l \rrbracket^{0'})$). La concaténation de vecteurs sera notée @. Enfin un vecteur v sera dit normalisé ssi la liste de blocs $\llbracket \phi(v) \rrbracket^0$ est normalisée.

4.2 Caractérisation de $\sqsubseteq$ sur les listes

4.2.1 Pré-ordre point par point modulo déformation du temps $\lesssim$

Intuition Un vecteur d'états correspond graphiquement à une courbe de consommation du programme en fonction du temps. Cette intuition inspire deux remarques :
- Étant donné que le temps n'a pas d'importance, deux courbes que l'on peut transformer l'une en l'autre par changement de coordonnées (strictement croissant) sur le temps, devraient être équivalentes.
- De plus, on peut s'attendre raisonnablement à ce qu'une courbe située en dessous d'une autre (inférieure point par point) lui soit inférieure pour $\sqsubseteq$.

Dès lors on peut se poser la question de savoir si ces deux faits sont suffisants pour expliquer entièrement $\sqsubseteq$.

Définition 4.2.1 (pré-ordre $\lesssim$). Soit $\rhd$ la relation de "suppression d'une répétition" sur les vecteurs :

$$[a_0\ \ldots\ a_{i-1} a_i a_i a_{i+1} \ldots a_n] \rhd [a_0\ \ldots\ a_{i-1} a_i a_{i+1} \ldots a_n]$$

et $\dot{\leq}$ le pré-ordre de comparaison "coordonnées par coordonnées" :

$$v_1 \dot{\leq} v_2 \text{ ssi } lth(v_1) = lth(v_2) \text{ et } \forall i,\ v_{1i} \leq v_{2i}$$

On définit le pré-ordre $\lesssim$ sur les vecteurs par :

$$v_1 \lesssim v_2 \text{ ssi } v_1 \overset{*}{\lhd} \dot{\leq} \overset{*}{\rhd} v_2$$

Théorème 4.2.2. *Pour tous v_1, v_2,*

$$v_1 \lesssim v_2 \Rightarrow v_1 \sqsubseteq v_2$$

Démonstration. On remarque d'une part que $(\rhd) \subseteq (\sqsupseteq)$ et d'autre part que $(\dot{\leq}) \subseteq (\sqsubseteq)$. $\square$

4.2.2 Présentation de $\lesssim$ en termes de chemins

Soit $v_1 = [v_{10}\, v_{11}\, \ldots\, v_{1n_1}]$ et $v_2 = [v_{20}\, v_{21}\, \ldots\, v_{2n_2}]$ deux vecteurs de tailles respectives n_1+1 et n_2+1, et supposons $v_1 \lesssim v_2$. Il existe donc v_1', v_2' de même taille, disons $n+1$, tels que $v_1 \overset{*}{\lhd} v_1' \overset{\cdot}{\leq} v_2' \overset{*}{\rhd} v_2$.

Une manière d'exprimer $v_i \overset{*}{\lhd} v_i'$ est de noter l'existence d'une fonction $\gamma_i : \{0 \ldots n\} \to \{0 \ldots n_i\}$ telle que :

$$\begin{cases} \forall t < n,\ \gamma_i(t) - \gamma_i(t-1) \in \{0,1\} \\ \forall t,\ v_{i\gamma_i(t)} = v_{it}' \end{cases}$$

L'hypothèse $v_1 \lesssim v_2$ s'écrit alors : $\forall t,\ v_{1\gamma_1(t)} \leq v_{2\gamma_2(t)}$.

Cette dernière équation s'écrit aussi :

$$\max_t \left(v_{1\gamma_1(t)} + (-v_{2\gamma_2(t)})\right) \leq 0$$

ce qui fait fortement penser à la définition du coût d'un chemin. Néanmoins si l'on pose $\gamma = \gamma_1 \times \gamma_2$, on a :

$$\forall t < n,\ \gamma(t+1) - \gamma(t) \in \{(0,0),(0,1),(1,0),(1,1)\}$$

ce qui n'est pas encore la forme voulue.

Il est facile d'éliminer le cas $\gamma(t+1)-\gamma(t) = (0,0)$ en construisant un nouveau chemin dans lequel ces étapes inutiles sont supprimées. Pour ce qui concerne le cas $\gamma(t+1)-\gamma(t) = (1,1)$, le lemme suivant montre qu'on peut toujours remplacer le déplacement $(1,1)$ par deux déplacements : ou bien $(0,1)(1,0)$ ou bien $(1,0)(0,1)$.

Lemme 4.2.3. *Soit $a_1, a_2, b_1, b_2 \in \mathbb{Z}$. On a :*

$$\max(a_1+a_2, b_1+b_2) \geq \min(\max(a_1+a_2, b_1+a_2, b_1+b_2), \max(a_1+a_2, a_1+b_2, b_1+b_2))$$

$$\begin{tikzcd} a_1+b_2 \arrow[r] & b_1+b_2 \\ a_1+a_2 \arrow[u] \arrow[r] \arrow[ur] & b_1+a_2 \arrow[u] \end{tikzcd}$$

Démonstration. Si l'on soustrait a_1+a_2 des membres de l'équation, et si l'on pose $x_1 = b_1 - a_1$ et $x_2 = b_2 - a_2$, l'équation à montrer devient :

$$\max(0, x_1+x_2) \geq \min(\max(0, x_1, x_1+x_2), \max(0, x_2, x_1+x_2))$$

Si x_1 et x_2 sont de même signe, ces quantités sont égales. Sinon, on a par exemple $x_1 \leq 0 \leq x_2$ ce qui implique :

$$\max(0, x_1+x_2) \geq \max(0, x_1, x_1+x_2)$$

□

Notons $-v \overset{déf}{=} [-v_i]_{0 \leq i \leq lth(v)-1}$. Tous ces éléments nous permettent maintenant d'établir la caractérisation suivante :

Théorème 4.2.4. *Pour tous v_1, v_2,*

$$v_1 \lesssim v_2 \Leftrightarrow \mathcal{C}(v_1 \| -v_2) \leq 0$$

4.2.3 Équivalence de $\lesssim$ et $\sqsubseteq$

La nouvelle définition de $\lesssim$ nous permet maintenant d'établir à peu de frais l'équivalence avec $\sqsubseteq$:

Théorème 4.2.5. *Soit v_1, v_2 des vecteurs. Les propositions suivantes sont équivalentes :*

1. $v_1 \lesssim v_2$
2. $\mathcal{C}(v_1 \| - v_2) \leq 0$
3. $v_1 \sqsubseteq v_2$

Démonstration. $1 \Leftrightarrow 2$ a été déjà prouvé, de même que $1 \Rightarrow 3$. Reste à montrer $3 \Rightarrow 2$. Or, sachant que

$$\forall w,\ \mathcal{C}(v_1 \| w) \leq \mathcal{C}(v_2 \| w)$$

on a en particulier :

$$\mathcal{C}(v_1 \| - v_2) \leq \mathcal{C}(v_2 \| - v_2) = 0$$

(on montre $\mathcal{C}(v_2 \| - v_2) = 0$ en revenant par exemple à la première définition de $\lesssim$) $\square$

Si $l = x_1 \dots x_n$ est une liste de variations, on pose : $-l \stackrel{déf}{=} (-x_1) \dots (-x_n)$. Dans ces conditions, on vérifie que $\phi(-l) = -\phi(l)$ ce qui nous permet de traduire les résultats précédents en terme de liste de variations :

Théorème 4.2.6. *Soit l_1, l_2 des listes de variations. On a l'équivalence :*

$$l_1 \sqsubseteq l_2 \Leftrightarrow \mathcal{C}(l_1 \| - l_2) \leq 0$$

Remarque. Ce théorème est aussi valable pour les listes de blocs, pour peu que l'on définisse un opérateur $(-)$ en revenant aux listes de variations :

$$-l \stackrel{déf}{=} [\![- [\![l]\!]^{0'}]\!]^{0}$$

En guise d'application, on peut énoncer le résultat suivant qui n'avait rien d'évident d'après la définition initiale de $\sqsubseteq$:

Corollaire 4.2.7. *Soit l_1, l_2 des listes de variations. On a :*

$$l_1 \sqsubseteq l_2 \Leftrightarrow -l_2 \sqsubseteq -l_1$$

On peut également établir la proposition suivante à propos des répétitions de programmes :

Proposition 4.2.8. *Pour tout processus p,*

$$\begin{aligned} \Delta(p) \leq 0 \ &\Rightarrow\ p\,p \sqsubseteq p \\ \Delta(p) \geq 0 \ &\Rightarrow\ p\,p \sqsupseteq p \\ \Delta(p) = 0 \ &\Rightarrow\ p\,p \sqsubseteq\!\!\!\sqsupseteq p \end{aligned}$$

Démonstration. Soit $l = [\![[\![p]\!]^2]\!]^{0'}$ une liste de variation équivalente à p. Découpons l selon un point où son coût est atteint : $l = l_1\, l_2$ avec $\Delta(l_1) = \mathcal{C}(l)$. On a plus précisément : $\mathcal{C}(l_1) = \Delta(l_1)$ et $\mathcal{C}(l_2) = 0$.

Si $\Delta(l) \leq 0$, on a alors :

$$\mathcal{C}(l\,l \| {-l}) \leq \mathcal{C}((l_1 \| {-l_1})\, l_2\, l_1\, (l_2 \| {-l_2})) = \mathcal{C}(l_2\, l_1) = \Delta(l) \leq 0$$

On conclut les autres cas en remarquant que si $\Delta(l) \geq 0$:

$$\Delta(-l) \leq 0 \ \Rightarrow\ -l{-l} \sqsubseteq -l \ \Rightarrow\ l\,l \sqsupseteq l$$

$\square$

4.3 Étude de $\sqsubseteq$ sur les listes normalisées

Soit l et l' deux listes de blocs normalisées, et a, a' deux entiers. Supposons $a + l \sqsubseteq a' + l'$. A l'aide des outils mis en place dans le paragraphe précédent, nous allons montrer que l'on a en fait $a = a'$ et $l = l'$.

Posons $v = a + l$ et $v' = a' + l'$. D'après le paragraphe précédent, $v \sqsubseteq v'$ implique d'une part $\mathcal{C}(v\| - v') \leq 0$ et d'autre part $\mathcal{C}(-v\|v') \leq 0$.

Soit $A = v \oplus (-v')$. Par définition, $\mathcal{C}(v\| - v') \leq 0$ signifie qu'il existe un chemin γ^- tel que $A_{\gamma^-} \leq 0$. Quant à l'équation $\mathcal{C}(-v\|v') \leq 0$, sachant que $(-v) \oplus v' = -A$, elle implique l'existence d'un chemin γ^+ tel que $A_{\gamma^+} \geq 0$.

Posons $v = [a_0 \ldots a_{2n}]$ et $v' = [a'_0 \ldots a'_{2n'}]$ (et donc $a_0 = a$ et $a'_0 = a'$). L'existence des chemins γ^+ et γ^- implique d'ores et déjà $a_0 = a'_0$.

Soit $m \in \{0 \ldots n\}$ le nombre de blocs de l ayant une variation $\delta_i < 0$. L'hypothèse de normalisation de v s'écrit :

$$\begin{cases} a_1 < a_3 < \cdots < a_{2m-1} < a_{2m+1} > \ldots > a_{2n-1} \\ a_0 > a_2 > \ldots > a_{2m} \leq a_{2m+2} < \ldots < a_{2n} \end{cases}$$

On dispose également des hypothèses de bonne formation des blocs :

$$\forall i < n,\ a_{2i} \leq a_{2i+1} \geq a_{2i+2}$$

qui sont équivalentes ici à $a_0 \leq a_1$ et $a_{2n-1} \geq a_{2n}$. Naturellement v' vérifie la même propriété pour un certain m'.

Pour montrer $v = v'$, on procède par récurrence sur $(n, n - m)$ (muni de l'ordre lexicographique) :

- Si $n = 0$ (i.e. $v = [a_0]$) : la matrice A est une "colonne". Il n'y a qu'un seul chemin à la fois positif et négatif, autrement dit nul, d'où par normalisation $v' = [a_0] = v$. Le même raisonnement s'applique si $n' = 0$.
- Si $n > 0$, $n' > 0$ et $m = 0$ alors les hypothèses sur v impliquent : $\forall i,\ a_i \geq a_0$. Pour qu'il existe un chemin γ^- il est nécessaire que : $\forall i,\ a'_i \geq a_0$ (le chemin passe par toutes les lignes de A). En particulier $a'_2 \geq a'_0$ et donc $m' = 0$.
  On peut alors se ramener au cas $m = n > 0$ et $m' = n' > 0$ par "symétrie sur le temps". On pose en effet : $w = [a_{2n} \ldots a_0]$ et $w' = [a'_{2n'} \ldots a'_0]$. Il est clair que $v \sqsubseteq v' \Leftrightarrow w \sqsubseteq w'$.
- Si $n, n', m, m' > 0$ et $a_1 = a_0$: on montre que $a'_1 = a'_0$ (et donc $a_1 = a'_1$) par l'absurde. En effet, si $a'_1 > a'_0$, le premier "carré" de la matrice A s'écrit :

$$\begin{array}{c|ccc} -a'_2 & > 0 & > 0 & < 0 \\ -a'_1 & < 0 & < 0 & < 0 \\ -a'_0 & = 0 & = 0 & < 0 \\ \hline & a_0 & a_1 & a_2 \end{array}$$

  ce qui contredit l'existence d'un chemin γ^+.
- Si $n, n', m, m' > 0$, $a_1 > a_0$ et $a'_1 > a'_0$, alors le premier "carré" de la matrice A est de la forme :

$$\begin{array}{c|ccc} -a'_2 & > 0 & > 0 & \\ -a'_1 & < 0 & ? & < 0 \\ -a'_0 & = 0 & > 0 & < 0 \\ \hline & a_0 & a_1 & a_2 \end{array}$$

On remarque que les deux chemins γ^+ et γ^- passent nécessairement par la case marquée '?'. Par conséquent $a_1 = a'_1$.

– Supposons maintenant $n, n', m, m' > 0$, $a'_1 = a_1$ en montrons que $[a_2 \ldots a_{2n}] \sqsubseteq [a'_2 \ldots a'_{2n'}]$.
En supposant[1] que $n > 1$ et $n' > 1$, et étant donné que $a_1 < a_3$ et $a'_1 < a'_3$, le début de la matrice A s'écrit maintenant :

$$\begin{array}{r|cccc}
-a'_3 & > 0 & > 0 & > 0 & \\
-a'_2 & > 0 & > 0 & ? & > 0 \\
-a'_1 & \leq 0 & = 0 & < 0 & > 0 \\
-a'_0 & = 0 & \geq 0 & < 0 & > 0 \\
\hline
 & a_0 & a_1 & a_2 & a_3
\end{array}$$

Les chemins γ^+ et γ^- sont obligés de passer par la case '?'; la suite du trajet des chemins nous donne par conséquent $[a_2 \ldots a_{2n}] \sqsubseteq [a'_2 \ldots a'_{2n'}]$ (noter que ces vecteurs sont encore normalisés).

Finalement on a montré :

Théorème 4.3.1. *Soit v_1 et v_2 des vecteurs normalisés.*

$$v_1 \sqsubseteq v_2 \Leftrightarrow v_1 = v_2$$

Corollaire 4.3.2. *Soit l_1 et l_2 des listes de blocs normalisées.*

$$l_1 \sqsubseteq l_2 \Leftrightarrow l_1 = l_2$$

4.4 Existence et calcul de la borne supérieure $\sqcup$

Nous montrons maintenant l'existence de la *borne supérieure*[2] de deux éléments, et donnons un algorithme pour la calculer.

Remarque. En vertu du corollaire 4.2.7, on déduira immédiatement de cet section l'existence et un algorithme de calcul de la *borne inférieure* :

$$v_1 \sqcap v_2 \stackrel{déf}{=} simpl^*(-((-v_1) \sqcup (-v_2)))$$

4.4.1 Étude de l'ensemble des majorants

Supposons $v_1 \sqsubseteq v$ et $v_2 \sqsubseteq v$ et posons $lth(v) = n + 1$, $lth(v_1) = n_1 + 1$, $lth(v_2) = n_2 + 1$. D'après le théorème 4.2.5, il existe deux chemins $(\alpha_1 \times \beta_1) : (0, 0) \rightsquigarrow (n_1, n)$ et $(\alpha_2 \times \beta_2) : (0, 0) \rightsquigarrow (n_2, n)$ tels que :

$$\begin{cases} \forall t_1 \in \{0 \ldots n_1 + n\},\ v_{1\alpha_1(t_1)} \leq v_{\beta_1(t_1)} \\ \forall t_2 \in \{0 \ldots n_2 + n\},\ v_{2\alpha_2(t_2)} \leq v_{\beta_2(t_2)} \end{cases}$$

β_1 et β_2 sont deux fonctions croissantes surjectives à valeurs dans $\{0 \ldots \mathrm{n}\}$. Il existe donc deux suites d'entiers $0 = t_0 < t_1 < \ldots < t_n < t_{n+1} = n_1 + n + 1$ et $0 = t'_0 <$

[1] Le même argument s'applique dans les autres cas.

[2] par *la* borne supérieure, nous désignons l'unique borne supérieure *normalisée* (*cf.* la section précédente).

$t'_1 < \ldots < t'_n < t'_{n+1} = n_2 + n + 1$ telles que : $\beta_1^{-1}(\{k\}) = \{t_k \ldots (t_{k+1} - 1)\}$ et $\beta_2^{-1}(\{k\}) = \{t'_k \ldots (t'_{k+1} - 1)\}$.

Soit $\delta = \max\{t_{k+1} - t_k, t'_{k+1} - t'_k\}$. On peut construire facilement deux fonctions $\phi_1 : \{0 \ldots \delta(n+1)\} \to \{0 \ldots n_1 + n\}$ et $\phi_2 : \{0 \ldots \delta(n+1)\} \to \{0 \ldots n_2 + n\}$) telles que :

$$\begin{cases} \forall k, & \phi_1(\delta k) = t_k \text{ et } \phi_2(\delta k) = t'_k \\ \forall t, k, & \delta k \leq t < \delta(k+1) \Longrightarrow t_k \leq \phi_1(t) < t_{k+1} \text{ et } t'_k \leq \phi_2(t) < t'_{k+1} \\ \forall t, & \phi_1(t+1) - \phi_1(t) \in \{0, 1\} \\ \forall t, & \phi_2(t+1) - \phi_2(t) \in \{0, 1\} \end{cases}$$

Dans ces conditions, on a alors : $\beta_1 \circ \phi_1 = \beta_2 \circ \phi_2$. Posons maintenant : $\gamma = (\alpha_1 \circ \phi_1) \times (\alpha_2 \circ \phi_2)$ et $M = [\max(v_{1i}, v_{2j})] \in \mathbb{Z}^{\{0 \ldots n_1\} \times \{0 \ldots n_2\}}$. γ est un chemin $(0,0) \rightsquigarrow (n_1, n_2)$ avec répétitions et “pas en diagonale”. Il vérifie :

$$\forall t, M_{\gamma(t)} \leq v_{\beta_1 \circ \phi_1(t)}$$

D'après la discussion précédente sur $\lesssim$, il existe donc un chemin normalisé $\gamma' : (0,0) \rightsquigarrow (n_1, n_2)$ tel que : $M_\gamma \sqsubseteq v$.

Théorème et définition 4.4.1. *Soit* $M = [\max(v_{1i}, v_{2j})] \in \mathbb{Z}^{\{0 \ldots n_1\} \times \{0 \ldots n_2\}}$. *Les vecteurs* M_γ *sont appelés* vecteurs candidats *du problème* $v_1 \sqcup v_2$. *Si l'on note* $v \uparrow$ *l'ensemble des majorants de* v*, ces vecteurs vérifient :*

$$\bigcup_\gamma (M_\gamma \uparrow) = (v_1 \uparrow) \cap (v_2 \uparrow)$$

4.4.2 Tests expérimentaux

Le raisonnement précédent permet de calculer les générateurs de $(v_1 \uparrow) \cap (v_2 \uparrow)$. Expérimentalement, nous avons cherché à les comparer entre eux (puisque l'on a maintenant un algorithme pour décider de $\sqsubseteq$) ; le résultat étonnant de nos “expériences” a été que :

1. Il existe toujours un minimum parmi les vecteurs candidats, et par conséquent une borne supérieure à v_1 et v_2.
2. Plus précisément un minorant des vecteurs candidats est systématiquement le vecteur associé au chemin optimal de l'algorithme d'ordonnancement.

4.4.3 Borne supérieure entre deux blocs

Ce résultat expérimental incite à étudier un lien entre $\preceq$ et $\sqcup$ en commençant par une paire de blocs.

Convenons de noter un chemin par la suite de ses étapes : le chiffre 1 pour une étape horizontale et 2 pour une étape verticale. Par exemple, 1122 désigne le chemin $(0,0)(1,0)(2,0)(2,1)(2,2)$.

Lemme 4.4.2. *Soit* (c_1, δ_1), (c_2, δ_2) *deux blocs, et* v_1, v_2 *deux vecteurs associés :* $v_i = [a_i, a_i + c_i, a_i + \delta_1]$. *Notons* $M = [\max(v_{1i}, v_{2j})] \in \mathbb{Z}^{\{0,1,2\} \times \{0,1,2\}}$

Si $(c_1, \delta_1) \preceq (c_2, \delta_2)$ *alors :*

$$\begin{aligned} v_1 \sqcup v_2 &= M_{1122} \\ &= [\max(a_1, a_2), \max(a_1 + c_1, a_2), \max(a_1 + \delta_1, a_2), \\ &\quad \max(a_1 + \delta_1, a_2), \max(a_1 + \delta_1, a_2 + c_2), \max(a_1 + \delta_1, a_2 + \delta_2)] \end{aligned}$$

Il y a $\binom{4}{2} = 6$ chemins possibles a priori. En fait, étant donné que $a_i \leq a_i + c_i \geq a_i + \delta_i$, on peut éliminer les quatre chemins passant par le centre : 1212, 1221, 2112 et 2121.[3] La propriété à montrer est donc :

$$(c_1, \delta_1) \preceq (c_2, \delta_2) \Rightarrow M_{1122} \sqsubseteq M_{2211}$$

Si l'on utilise la caractérisation précédente, il faut montrer l'existence d'un chemin négatif dans la matrice $(M_{1122} \oplus (-M_{2211}))$ qui est de taille 5×5, ce qui fait a priori $\binom{8}{4} = 70$ chemins possibles.

Sous cette forme la propriété a été vérifié formellement par Lilian Burdy à l'aide de l'atelier B, commercialisé par la société ClearSy [Cle]. On trouvera la spécification en B à l'annexe C.

4.4.4 Calcul de $\sqcup$ entre deux listes triées

À la manière de la démonstration du lemme 2.4.2, on généralise maintenant ce résultat à des listes triées.

Lemme 4.4.3. *Soit $l_1 = y_1\, l'_1$, $l_2 = y_2\, l'_2$ deux listes de blocs triées de tailles n_1 et n_2 (> 0 donc). Soit a_1 et a_2 deux entiers. Posons $y_i = (c_i, \delta_i)$. Si $y_1 \preceq y_2$ et si $(a_1 + \delta_1 + l'_1) \sqcup (a_2 + l_2)$ existe, alors :*

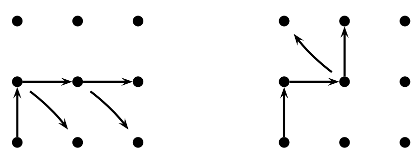
$$(a_1 + l_1) \sqcup (a_2 + l_2) \sqsupseteq [\max(a_1, a_2), \max(a_1 + c_1, a_2)]@((a_1 + \delta_1 + l'_1) \sqcup (a_2 + l_2))$$

Démonstration. Tout d'abord on peut noter que le membre droit est bien un majorant : c'est même le vecteur candidat correspondant au chemin 11 suivie du chemin γ_0 associé à $(a_1 + \delta_1 + l'_1) \sqcup (a_2 + l_2)$.

Notons $M = [\max((a_1 + l_1)_i, (a_2 + l_2)_j] \in \mathbb{Z}^{\{0\ldots 2n_1\}\times\{0\ldots 2n_2\}}$ et soit γ un chemin quelconque : $(0,0) \rightsquigarrow (2n_1, 2n_2)$.

Montrons tout d'abord qu'on peut se ramener au cas où le chemin γ ne passe par aucun centre des carrés "bloc fois bloc", c'est-à-dire par aucun sommet $(2i+1, 2j+1)$.

Si c'est le cas, en effet, considérons le premier point $(2i+1, 2j+1)$ rencontré. Soit $[a_i, a_i + c_i, a_i + \delta_i]$ les deux vecteurs correspondants. Le carré associé est nécessairement d'une des formes suivantes (modulo symétrie par rapport à la diagonale) :



Sachant que $a_i \leq a_i + c_i \geq a_i + \delta_i$, on peut déformer le chemin dans les deux cas (voir les flèches) de manière à obtenir un vecteur candidat plus petit pour $\sqsubseteq$ et comportant un "centre de carré" de moins. En effet :

$$\begin{aligned} & [\max(a_1, a_2), \max(a_1, a_2 + c_2), \max(a_1 + c_1, a_2 + c_2), \max(a_1 + \delta_1, a_2 + c_2)] \\ \sqsupseteq\ & [\max(a_1, a_2), \max(a_1 + c_1, a_2), \max(a_1 + \delta_1, a_2), \max(a_1 + \delta_1, a_2 + c_2)] \end{aligned}$$

et

$$\begin{aligned} & [\max(a_1, a_2), \max(a_1, a_2 + c_2), \max(a_1 + c_1, a_2 + c_2), \max(a_1 + c_1, a_2 + \delta_2)] \\ \sqsupseteq\ & [\max(a_1, a_2), \max(a_1, a_2 + c_2), \max(a_1, a_2 + \delta_2), \max(a_1 + c_1, a_2 + \delta_2)] \end{aligned}$$

[3] l'argument complet serait détaillé dans la preuve du théorème 4.4.3

On peut donc supposer γ compatible avec les blocs. Si γ commence par 11, alors la suite de γ correspond à un majorant de $a_1 + \delta_1 + l'_1$ et de $a_2 + l_2$. Et par conséquent :

$$M_\gamma \sqsupseteq [\max(a_1, a_2), \max(a_1 + c_1, a_2)]@((a_1 + \delta_1 + l'_1) \sqcup (a_2 + l_2))$$

Dans le cas contraire, γ commence par un chemin de la forme $\underbrace{2\ldots2}_{2m \text{ fois}}11$, avec $m > 0$. Les 4 dernières étapes de ce chemin sont inscrites dans le carré correspondant aux blocs y_1 et $y_{2,m}$, où $y_{2,m}$ est le m-ième blocs de l_2. Comme $y_1 \preceq y_2$ et que l_2 est triée, on a $y_1 \preceq y_{2,m}$. Le lemme 4.4.2 s'applique et montre que le vecteur candidat $\gamma' \stackrel{déf}{=} \underbrace{2\ldots2}_{2m-2 \text{ fois}}1122$ vérifie $\gamma' \sqsubseteq \gamma$. Par récurrence, on se ramène donc au premier cas. □

Quant au cas de base de la récurrence, il est traité par le théorème suivant :

Lemme 4.4.4. *Soit $v = [a_0\, a_1\, \ldots\, a_n]$ un vecteur et $a \in \mathbb{Z}$. On a :*

$$[a] \sqcup v = [\max(a, a_0)\ \max(a, a_1)\ \ldots\ \max(a, a_n)]$$

Démonstration. Il n'y a qu'un seul vecteur candidat à considérer. □

Finalement, les deux lemmes précedents permettent de montrer le résultat escompté :

Théorème 4.4.5 (Existence et calcul de $\sqcup$). *Soit v_1 et v_2 deux vecteurs de taille n_1+1 et n_2+1. La borne supérieure de v_1 et de v_2 existe, et est calculée par l'algorithme suivant de complexité $O(n_1+n_2)$:*
- *On pose $v_i = a_i + l_i^0$ où l_i^0 est une liste de blocs.*
- *Soit $l_i = simpl^*(l_i^0)$ la liste normalisée correspondant à l_i^0.*
- *On calcule récursivement $(a_1 + l_1) \sqcup (a_2 + l_2)$ par :*
  - *Si $l_1 = \epsilon$ et $l_2 = \epsilon$, alors :*

    $$(a_1 + l_1) \sqcup (a_2 + l_2) = [\max(a_1, a_2)]$$

  - *Si ($l_1 = y_1\, l'_1$ et $l_2 = \epsilon$) ou si ($l_1 = y_1\, l'_1$, $l_2 = y_2\, l'_2$ et $y_1 \preceq y_2$), alors :*

    $$(a_1+l_1)\sqcup(a_2+l_2) = [\max(a_1, a_2), \max(a_1+c_1, a_2)]@((a_1+\delta_1+l'_1)\sqcup(a_2+l_2))$$

  - *Si ($l_1 = \epsilon$ et $l_2 = y_2\, l'_2$) ou si ($l_1 = y_1\, l'_1$, $l_2 = y_2\, l'_2$ et $y_1 \succeq y_2$), alors :*

    $$(a_1+l_1)\sqcup(a_2+l_2) = [\max(a_1, a_2), \max(a_1, a_2+c_2]@((a_1+l_1)\sqcup(a_2+\delta_2+l'_2))$$

- *Pour finir, on normalise le résultat obtenu.*

Finalement, les listes forment un treillis dont toutes les opérations usuelles sont calculables en temps linéaire.

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

À la recherche d'un critère de détection des états dangereux, nous avons introduit un langage de processus, puis démontré l'existence d'un algorithme polynomial pour calculer le coût d'un état modélisé par un terme du langage.

La particularité de notre algorithme est de reposer sur une traduction compositionnelle des programmes vers des listes de variations normalisées. On peut ainsi analyser chaque programme séparément, et en annoter les endroits clés par des listes. À l'exécution les informations statiques seront rassemblées pour déterminer le coût de l'état courant.

Dans le but d'analyser des programmes réalistes il resterait à étendre le formalisme aux structures de contrôle. Néanmoins une étude plus approfondie des listes de variations nous a d'ores et déjà permis de montrer que les listes normalisées (et donc les processus) forment un treillis.

5.2 Travaux futurs

Dans ce qui suit, on discute d'un certain nombre d'extensions à apporter au formalisme pour pouvoir inclure des structures de contrôle.

5.2.1 Branchements

Une manière de simuler les branchements serait d'étendre le langage par un opérateur de *choix non-déterministe* :

$$\begin{array}{rcl} p & ::= & \dots \\ & | & (p_1 + p_2) \end{array}$$

Étant donné la structure de treillis des listes, une traduction raisonnable de l'opérateur $+$ est :

$$[\![p_1 + p_2]\!] = [\![p_1]\!] \sqcup [\![p_2]\!]$$

Malheureusement, pour montrer la correction de cette traduction, il faut a priori définir la sémantique du nouveau langage, ce qui s'avère relativement compliqué. Pour définir le coût d'un programme, on doit par exemple tenir compte de la présence de deux "joueurs" : l'ordonnanceur – qui choisit quel côté activer dans $p_1 \| p_2$ – et un "adversaire" – qui choisit si $p_1 + p_2 \to p_1$ ou si $p_1 + p_2 \to p_2$. La formule de définition est du coût est alors intuitivement une alternance de min et de max.

5.2.2 Boucles

On peut imaginer représenter les boucles par un opérateur de *répétition finie* (non-déterministe) :

$$\begin{array}{rcl} p & ::= & \ldots \\ & | & p* \end{array}$$

Si $\Delta(p) \leq 0$, étant donné la proposition 4.2.8 sur les répétitions de terme, on a $p^n \sqsubseteq p$ et par conséquent $\epsilon \sqcup p \sqcup \ldots \sqcup p^n \sqcup \ldots \sqsubseteq \epsilon \sqcup p$. Si $\Delta(p) > 0$, il est clair que le variation totale de p^n tend vers l'infini. Une traduction intuitive des boucles serait donc :

$$[\![p^*]\!] = \begin{cases} \epsilon \sqcup [\![p]\!] & \text{si } \Delta(p) \leq 0 \\ \top & \text{sinon} \end{cases}$$

Remarque. En pratique, on aura souvent à traduire un terme de la forme p^*l où l est la liste déjà obtenue par l'analyse statique de la suite du programme. Dans ce cas, il est plus précis de poser :

$$[\![p^*l]\!] = \begin{cases} l \sqcup [\![p\,l]\!] & \text{si } \Delta(p) \leq 0 \\ \top & \text{sinon} \end{cases}$$

5.2.3 Non-terminaison

Jusqu'à présent, les programmes étaient considérés comme terminant toujours en un temps borné, ce afin d'éviter la situation de *famine*. Malheureusement cette hypothèse est difficile à maintenir, ne serait-ce que du fait des boucles ou des entrées-sorties.

Pour modéliser cet état de fait, on ajoute un symbole pour modéliser un temps d'attente potentiellement infini :

$$\begin{array}{rcl} p & ::= & \ldots \\ & | & \uparrow \end{array}$$

Considérons en guise d'exemple les listes de variations $l_1 = (1 \ \uparrow \ -1)$ et $l_2 = 1$ en parallèle. Si l'analyse ignore l'opération $\uparrow$, elle évaluera le coût total à :

$$\mathcal{C}(l_1 \| l_2) = \mathcal{C}(1 \ -1 \ 1) = 1$$

Or pour une telle quantité de mémoire, le système doit forcément exécuter l_1 avant l_2, de sorte que l_2 attend un temps potentiellement infini.

Intuitivement, le problème vient ici du fait que le système comptait sur la promesse de libération de l_1 alors qu'elle n'était peut-être pas atteignable en un temps fini. Compte-tenu du paragraphe précédent sur les branchements, une manière naturelle de modéliser le fait qu'après un $\uparrow$, la suite du programme n'est peut-être pas accessible, serait de considérer $\uparrow p$ comme étant équivalent à $p + \epsilon$. Dans ces conditions, une traduction raisonnable des programmes en listes serait donnée par la fonction α (dont la seconde coordonnée représente la terminaison du programme) :

$$\begin{array}{rcll} \alpha(\epsilon) & = & (\epsilon, 0) & \\ \alpha(x\,p) & = & (simpl^*(x\,l), a) & \text{si } \alpha(p) = (l, a) \\ \alpha(\uparrow p) & = & (\epsilon \sqcup l, 1) & \text{si } \alpha(p) = (l, a) \\ \alpha((p_1 \| \ldots \| p_n)\,p) & = & \alpha(merge_{\preceq}(l_1, \ldots, l_n)\,p) & \text{si } \alpha(p_i) = (l_i, a_i) \text{ et } \forall i,\ a_i = 0 \\ \alpha((p_1 \| \ldots \| p_n)\,p) & = & \alpha(merge_{\preceq}(l_1, \ldots, l_n) \uparrow p) & \text{si } \alpha(p_i) = (l_i, a_i) \text{ et } \exists i,\ a_i = 1 \end{array}$$

Remarque.

1. L'union avec ϵ a pour effet de tronquer la partie négative du graphe de consommation : autrement dit on ne peut plus descendre en dessous du point de départ. Cet élément correspond à l'intuition précédente comme quoi si on ne peut pas prétendre dépasser le point d'attente $\uparrow$ un temps fini, il vaut mieux oublier les promesses de libérer de la mémoire.
2. La dernière remarque donne aussi la limite de cette correction : s'il y a trop d'attente infinie, les promesses de libération risquent d'être toutes oubliées, et finalement l'analyse concluera que le processus a besoin de son maximum théorique en permanence (c'est-à-dire que le coût total sera la somme des coûts).

5.2.4 Un prototype plus réaliste

Les extensions proposées pour manipuler les boucles et les branchements, même si elles n'ont pas encore été prouvées formellement, ont été expérimentées avec succès dans un nouveau prototype sur un petit langage impératif avec boucles et branchements conditionnels.

L'analyse des valeurs repose sur un interpréteur abstrait dans le treillis des intervalles. Les annotations sont calculées par un parcours arrière du graphe de flot, et inscrites comme argument supplémentaire aux allocations.

On trouvera un exemple d'exécution du simulateur en annexe à la section A.3.2.

5.3 Limites de la démarche

Une limitation visible de notre démarche est de ne considérer qu'une seule ressource. [Gol78] montre que dans le cas général, le problème de la prédiction des interblocages est NP-complet. En fait, même si on s'accomode d'un algorithme approché, notre étude se généralise difficilement à plusieurs ressources, car en présence d'ordres partiels la notion d'ordonnancement optimal disparaît.

L'hypothèse de terminaison de tous les programmes est une hypothèse assez contraignante en pratique. Nous avons proposé parmi les travaux futurs une manière d'intégrer dans le formalisme le cas des programmes qui ne terminent pas en temps fini. Malheureusement, ces corrections risquent de dégrader notablement les prévisions du système, car celui-ci devra éliminer un grand nombre d'ordonnancements de son calcul.

Enfin même si tous les programmes terminent en temps borné, nous ne nous sommes pas intéressés à l'équité de l'ordonnancement : l'utilisateur n'a donc aucune garantie sur la qualité du multiplexage des tâches. Une telle garantie aurait pourtant son intérêt dans un système temps-réél.

Annexe A

Exemples et résultats

A.1 Exemples de calcul de coût

Considérons les suites de variations $p_1 = (2\ 1\ {-2}\ 3\ {-4})$ et $p_2 = (1\ 1\ 1\ {-3})$ et calculons $\mathcal{C}(p_1 \parallel p_2)$ selon différents algorithmes.

A.1.1 Algorithme par exploration des états

Par commodité, on présente ici une version "absolue" de l'algorithme évoqué au chapitre 2.

Les consommations absolues respectives sont alors $v_1 = [0\ 2\ 3\ 1\ 4\ 0]$ et $v_2 = [0\ 1\ 2\ 3\ 0]$, de sorte que la matrice $A = (a_{i,j}) \stackrel{déf}{=} (v_{1i} + v_{2j})$ des consommations instantanées en fonction de l'avancement des tâches s'écrit :

$$
\begin{array}{cccccc}
0 & 2 & 3 & 1 & 4 & 0 \\
3 & 5 & 6 & 4 & 7 & 3 \\
2 & 4 & 5 & 3 & 6 & 2 \\
1 & 3 & 4 & 2 & 5 & 1 \\
0 & 2 & 3 & 1 & 4 & 0
\end{array}
$$

La matrice $B = (b_{i,j})$ des consommations "prévisionnelles" se calcule de proche en proche en partant du point d'arrivée selon la formule :

$$b_{i,j} = \max(a_{i,j}, \min(b_{i+1,j}, b_{i,j+1}))$$

(avec la convention $b_{i,j} = +\infty$ en dehors de la matrice).

On obtient ainsi :

$$
\begin{array}{cccccc}
4 & 4 & 4 & 4 & 4 & 0 \\
4 & 5 & 6 & 4 & 7 & 3 \\
4 & 5 & 5 & 4 & 6 & 3 \\
4 & 4 & 4 & 4 & 5 & 3 \\
4 & 4 & 4 & 4 & 4 & 3
\end{array}
$$

Ainsi le coût minimum pour mener à bien les tâches $p_1 = (2\ 1\ -2\ 3\ -4)$ et $p_2 = (1\ 1\ 1\ -3)$ est 4. Les autres cases donnent également les coûts pour terminer les tâches en cours (à condition de soustraire la valeur instantanée). Par exemple, le coût minimum pour terminer $(1\ -2\ 3\ -4)$ et $(1\ 1\ -3)$ est $4 - 3 = 1$.

A.1.2 Algorithme 1

En pratique, les tâches sont analysées en partant de la fin. On calcule directement la traduction finale $[\![\_]\!]^{01} \stackrel{déf}{=} [\![\ [\![\_]\!]^0]\!]^1$:

$$
\begin{array}{lcll}
[\![-4]\!]^{01} & = & (0,-4) & \\
[\![3\ -4]\!]^{01} & = & fusion((3,3)(0,-4)) & = (3,-1) \\
[\![-2\ 3\ -4]\!]^{01} & = & fusion((0,-2)(3,-1)) & = (1,-3) \\
[\![1\ -2\ 3\ -4]\!]^{01} & = & fusion((1,1)(1,-3)) & = (2,-2) \\
[\![p_1]\!]^{01} & = & fusion((2,2)(2,-2)) & = (4,0) \\
\\
[\![-3]\!]^{01} & = & (0,-3) & \\
[\![1\ -3]\!]^{01} & = & fusion((1,1)(0,-3)) & = (1,-2) \\
[\![1\ 1\ -3]\!]^{01} & = & fusion((1,1)(1,-2)) & = (2,-1) \\
[\![p_2]\!]^{01} & = & fusion((1,1)(2,-1)) & = (3,0)
\end{array}
$$

On en déduit :

$$[\![p_1 \parallel p_2]\!]^{01} = fusion^*(sort_{\preceq}((4,0)(3,0))) = fusion^*((4,0)(3,0) = (4,0)$$

Et de même :

$$[\![(1\ -2\ 3\ -4) \parallel (1\ 1\ -3)]\!]^{01} = fusion^*(sort_{\preceq}((2,-2)(2,-1))) = fusion^*((2,-2)(2,-1)) = (2,-3)$$

A.1.3 Algorithme 2

Sur l'exemple précédent, l'algorithme 2 donne le calcul suivant :

$$
\begin{array}{lcll}
[\![-4]\!]^{02} & = & (0,-4) & \\
[\![3\ -4]\!]^{02} & = & simpl^*((3,3)(0,-4)) & = (3,-1) \\
[\![-2\ 3\ -4]\!]^{02} & = & simpl^*((0,-2)(3,-1)) & = (0,-2)(3,-1) \\
[\![1\ -2\ 3\ -4]\!]^{02} & = & simpl^*((1,1)(0,-2)(3,-1)) & = (1,-1)(3,-1) \\
[\![p_1]\!]^{02} & = & simpl^*((2,2)(1,-1)(3,-1)) & = (4,0) \\
\\
[\![-3]\!]^{02} & = & (0,-3) & \\
[\![1\ -3]\!]^{02} & = & simpl^*((1,1)(0,-3)) & = (1,-2) \\
[\![1\ 1\ -3]\!]^{02} & = & simpl^*((1,1)(1,-2)) & = (2,-1) \\
[\![p_2]\!]^{02} & = & simpl^*((1,1)(2,-1)) & = (3,0)
\end{array}
$$

Par conséquent :

$$[\![p_1 \parallel p_2]\!]^{02} = simpl^*(merge_{\preceq}((4,0),(3,0))) = (4,0)$$

$$[\![(1\ -2\ 3\ -4) \parallel (1\ 1\ -3)]\!]^{02} = simpl^*(merge_{\preceq}((1,-1)(3,-1)(2,-1))) = (1,-1)(2,-1)(3,-1)$$

soit un coût de 1 (qui est visiblement la valeur optimale).

A.2 Étude expérimentale de la normalisation

La normalisation a pour effet de réduire la plupart des listes de manière notable, comme on peut le voir sur l'exemple suivant :

$$
\begin{array}{lcl}
t & = & (-6\ -4\ -9\ 8\ 5\ -7\ -7\ 5\ -1\ 6) \\
[\![t]\!]^{0} & = & (0,-6)(0,-4)(0,-9)(8,8)(5,5)(0,-7)(0,-7)(5,5)(0,-1)(6,6) \\
[\![t]\!]^{03} & = & (0,-6)(0,-4)(0,-9)(13,-1)(5,4)(6,6) \\
[\![t]\!]^{02} & = & (0,-20)(10,10)
\end{array}
$$

Pour se faire une idée du comportement statistique de la normalisation, on a étudié expérimentalement la longueur après normalisation sur des échantillons de 1000 listes tirées au sort. Le tirage au sort consistait à générer un vecteur de taille $n+1$ dont les coordonnées étaient choisies uniformément entre $-m$ et $+m$, puis à traduire ce vecteur en n blocs élémentaires.

Pour $m = 100$, le critère "minimal" de simplification $\mathcal{S}'$ a donné :

valeur de n	minimum	moyenne	maximum
10	1	3.43	8
100	2	6.84	12
1000	9	21.78	40
10000	131	171.04	214

Pour $m = 100$, le critère optimal de simplification $\mathcal{S}$ a donné :

valeur de n	minimum	moyenne	maximum
10	1	2.47	5
100	1	4.21	9
1000	1	4.43	9
10000	1	4.41	9

À m constant, la taille moyenne des listes normalisées tend donc expérimentalement vers une limite extrêmement basse.

Ce résultat dépend évidemment du modèle de tirage au sort des listes, mais il laisse à penser que la taille des listes manipulées dépassera rarement la dizaine de blocs en pratique.

A.3 Exemples d'exécution du prototype

Dans ces exemples, les processus sont implémentés par comme des processus légers de Ocaml. L'ordonnanceur de librairie Thread (non-native) de Ocaml n'étant pas préemptif, les processus rendent la main – par un 'yield' – après chaque allocation.

Lorsqu'un processus est gelé, le message "avoiding deadlock" signifie qu'une zone dangereuse[1] vient d'être évitée (ouf). Le message "not enough memory" signifie que l'allocation était de toute façon impossible par manque de mémoire.

[1] En fait, si les programmes contiennent des branchements c'est une zone *potentiellement* dangereuse.

A.3.1 Cas séquentiel

On reproduit ci-dessous la trace de l'exécution concurrente des trois listes :

$$\begin{aligned} t_1 &= (1\ 1\ 1\ -1\ 2\ -3) \\ t_2 &= (2\ 2\ -2\ 1\ -2) \\ t_3 &= (1\ 1\ -1\ 1\ 2\ -4) \end{aligned}$$

avec une quantité de mémoire de 5 (c'est-à-dire le minimum possible).

```
simul -a 2 -v 1 -f test_seq1.txt
starting simulation...

starting server (available memory : 5)
annotations for t1:
start [(4,1)]
alloc 1, [(3,0)]
alloc 1, [(2,-1)]
alloc 1, [(0,-1); (2,-1)]
alloc -1, [(2,-1)]
alloc 2, [(0,-3)]
alloc -3, []
end
annotations for t2:
start [(4,1)]
alloc 2, [(2,-1)]
alloc 2, [(0,-3)]
alloc -2, [(1,-1)]
alloc 1, [(0,-2)]
alloc -2, []
end
annotations for t3:
start [(4,0)]
alloc 1, [(3,-1)]
alloc 1, [(0,-1); (3,-1)]
alloc -1, [(3,-1)]
alloc 1, [(2,-2)]
alloc 2, [(0,-4)]
alloc -4, []
end
starting threads
t1 started as <1>
t2 started as <2>
t3 started as <3>
t1: alloc(1, [(3,0)])
thread <2> frozen: avoiding deadlock
t3: alloc(1, [(3,-1)])
t1: alloc(1, [(2,-1)])
thread <2> frozen: avoiding deadlock
t3: alloc(1, [(0,-1); (3,-1)])
t1: alloc(1, [(0,-1); (2,-1)])
thread <2> frozen: not enough memory
t3: alloc(-1, [(3,-1)])
t1: alloc(-1, [(2,-1)])
```

```
thread <2> frozen: avoiding deadlock
thread <3> frozen: avoiding deadlock
t1: alloc(2, [(0,-3)])
thread <2> frozen: not enough memory
thread <3> frozen: not enough memory
t1: alloc(-3, [])
thread <2> frozen: avoiding deadlock
t3: alloc(1, [(2,-2)])
t1 finished
thread <2> frozen: avoiding deadlock
t3: alloc(2, [(0,-4)])
thread <2> frozen: not enough memory
t3: alloc(-4, [])
t2: alloc(2, [(2,-1)])
t3 finished
t2: alloc(2, [(0,-3)])
t2: alloc(-2, [(1,-1)])
t2: alloc(1, [(0,-2)])
t2: alloc(-2, [])
t2 finished

report:
10 freezes for 23 allocations
maximal number of frozen threads: 2
```

A.3.2 Petit langage impératif avec branchements et boucles

On donne en entrée au simulateur la quantité de mémoire disponible et la description des programmes :

```
/* available memory: */
6

/* tasks: */
run {
  alloc(2);
  while (1 <= random_int(2)) {
    alloc(1);
    alloc(-2);
    alloc(4);
    alloc(-3);
  }
  alloc(-1);
}

run {
  alloc(2);
  if (1 <= random_int(2)) {
    alloc(1);
    alloc(-2);
    alloc(4);
    alloc(-5);
  } else {
    alloc(2);
```

```
    alloc(-3);
    alloc(4);
    alloc(-5);
    alloc(6);
    alloc(-7);
  }
}
```

On obtient alors (par exemple) la trace suivante :

```
starting simulation...

starting server (available memory : 6)
annotations for t1:
run [(5,1)] {
  /* env:[], a:[(5,1)] */
  alloc(2, [(1,-1); (4,0)]);
  /* env:[], a:[(1,-1); (4,0)] */
  while (1 <= random_int(2)) {
    /* env:[], a:[(1,-1); (4,0)] */
    alloc(1, [(0,-2); (4,0)]);
    /* env:[], a:[(0,-2); (4,0)] */
    alloc(-2, [(4,0)]);
    /* env:[], a:[(4,0)] */
    alloc(4, [(0,-4)]);
    /* env:[], a:[(0,-4)] */
    alloc(-3, [(1,-1); (4,0)]);
    /* env:[], a:[(1,-1); (4,0)] */
    }
  /* env:[], a:[(0,-1)] */
  alloc(-1, []);
  /* env:[], a:[] */
  }
annotations for t2:
run [(6,0)] {
  /* env:[], a:[(6,0)] */
  alloc(2, [(2,-1); (4,-1); (6,0)]);
  /* env:[], a:[(2,-1); (4,-1); (6,0)] */
  if (1 <= random_int(2)) {
    /* env:[], a:[(1,-1); (4,-1)] */
    alloc(1, [(0,-2); (4,-1)]);
    /* env:[], a:[(0,-2); (4,-1)] */
    alloc(-2, [(4,-1)]);
    /* env:[], a:[(4,-1)] */
    alloc(4, [(0,-5)]);
    /* env:[], a:[(0,-5)] */
    alloc(-5, []);
    /* env:[], a:[] */
    } else {
    /* env:[], a:[(2,-1); (4,-1); (6,-1)] */
    alloc(2, [(0,-3); (4,-1); (6,-1)]);
    /* env:[], a:[(0,-3); (4,-1); (6,-1)] */
    alloc(-3, [(4,-1); (6,-1)]);
    /* env:[], a:[(4,-1); (6,-1)] */
    alloc(4, [(0,-5); (6,-1)]);
```

```
    /* env:[], a:[(0,-5); (6,-1)] */
    alloc(-5, [(6,-1)]);
    /* env:[], a:[(6,-1)] */
    alloc(6, [(0,-7)]);
    /* env:[], a:[(0,-7)] */
    alloc(-7, []);
    /* env:[], a:[] */
    }
  /* env:[], a:[] */
  }
starting threads
t1 started as <1>
t2 started as <2>
thread <1> frozen: avoiding deadlock
t2: alloc(2, [(2,-1); (4,-1); (6,0)])
thread <1> frozen: avoiding deadlock
t2: alloc(1, [(0,-2); (4,-1)])
t1: alloc(2, [(1,-1); (4,0)])
t2: alloc(-2, [(4,-1)])
t1: alloc(1, [(0,-2); (4,0)])
thread <2> frozen: not enough memory
t1: alloc(-2, [(4,0)])
t2: alloc(4, [(0,-5)])
thread <1> frozen: not enough memory
t2: alloc(-5, [])
t1: alloc(4, [(0,-4)])
t2 finished
t1: alloc(-3, [(1,-1); (4,0)])
t1: alloc(-1, [])
t1 finished

report:
4 freezes for 15 allocations
maximal number of frozen threads: 1
```

Annexe B

Compléments

Proposition B.0.1 (Compatibilité de la fusion avec $\preceq$).
Pour tout $\mathcal{R} \in \{\preceq, \succeq, \prec, \succ\}$*, pour tous* $(c_1, \delta_1), (c_2, \delta_2), (c_3, \delta_3)$*, on a :*

$$\left\{ \begin{array}{ll} (c_1, \delta_1)\mathcal{R}(c_2, \delta_2) & (H_1) \\ (c_1, \delta_1)\mathcal{R}(c_3, \delta_3) & (H_2) \end{array} \right. \Longrightarrow (c_1, \delta_1)\mathcal{R}(\max(c_2, \delta_2 + c_3), \delta_2 + \delta_3)$$

Ce résultats se montre de manière assez laborieuse mais systématique en distinguant tous les cas possibles dans les hypothèses.

Proposition B.0.2. $\xrightarrow{simpl'}$ *est confluente.*

Démonstration. L'unique paire critique s'écrit :

$$\begin{array}{ccc} (c_1, \delta_1)(c_2, \delta_2)(c_3, \delta_3) & \longrightarrow & (\max(c_1, \delta_1 + c_2), \delta_1 + \delta_2)(c_3, \delta_3) \\ \big\downarrow & & \big\downarrow ? \\ (c_1, \delta_1)(\max(c_2, \delta_2 + c_3), \delta_2 + \delta_3) & \xrightarrow{?} & (\max(c_1, \delta_1 + c_2, \delta_1 + \delta_2 + c_3), \delta_1 + \delta_2 + \delta_3) \end{array}$$

Il suffit donc de vérifier :

$$\left\{ \begin{array}{l} (c_1, \delta_1) \succ (c_2, \delta_2) \\ (c_2, \delta_2) \succ (c_3, \delta_3) \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} (\max(c_1, \delta_1 + c_2), \delta_1 + \delta_2) \succ (c_3, \delta_3) \\ (c_1, \delta_1) \succ (\max(c_2, \delta_2 + c_3), \delta_2 + \delta_3) \end{array} \right.$$

Or c'est une conséquence de la proposition B.0.1. $\square$

Annexe C

Énoncé du lemme 4.4.2 en B

```
MACHINE
    preuve

ABSTRACT_CONSTANTS
    d0, c1, d1, c2, d2,
    aa, bb, cc, dd, ee, ff, gg, hh,
    a11, a12, a13, a14, a15,

...

PROPERTIES
    c1 : INTEGER & c2 : INTEGER & d0 : INTEGER & d1 : INTEGER & d2 : INTEGER &
    aa : INTEGER & bb : INTEGER & cc : INTEGER & dd : INTEGER &
    ee : INTEGER & ff : INTEGER & gg : INTEGER & hh : INTEGER &

    a11 : BOOL & a12 : BOOL & a13 : BOOL & a14 : BOOL & a15 : BOOL &

...

ASSERTIONS
(   c1 >= 0 & c1 >= d1 & c2 >= 0 & c2 >= d2 &

    ((d1 <= 0 & d2 >= 0) or
     (d1 < 0 & d2 < 0 & c1 <= c2) or
     (d1 > 0 & d2 > 0 & c2 + d1 <= c1 + d2)) &

    ((0 <= d0 & aa = d0) or (d0 <= 0 & aa =0)) &
    ((0 <= d0 + c1 & bb = d0+c1) or (0 >= d0 + c1 & bb = 0)) &
    ((0 <= d0+d1 & cc = d0+d1) or (0 >= d0+d1 & cc = 0)) &
    ((c2 <= d0+d1 & dd = d0+d1) or (c2 >= d0+d1 & dd = c2)) &
    ((c2 <= d0 & ee = d0) or (c2 >= d0 & ee = c2)) &
    ((d2 <= d0 & ff = d0) or (d2 >= d0 & ff = d2)) &
    ((d2 <= d0+c1 & gg = d0+c1) or (d2 >= d0+c1 & gg = d2)) &
    ((d2 <= d0+d1 & hh = d0+d1) or (d2 >= d0+d1 & hh = d2)) &

    a11 = bool(aa <= aa) &
    a12 = bool(aa <= ee) &
    a13 = bool(aa <= ff) &
    a14 = bool(aa <= gg) &
```

```
    a15 = bool(aa <= hh) &
    a21 = bool(bb <= aa) &
    a22 = bool(bb <= ee) &
    a23 = bool(bb <= ff) &
    a24 = bool(bb <= gg) &
    a25 = bool(bb <= hh) &
    a31 = bool(cc <= aa) &
    a32 = bool(cc <= ee) &
    a33 = bool(cc <= ff) &
    a34 = bool(cc <= gg) &
    a35 = bool(cc <= hh) &
    a41 = bool(dd <= aa) &
    a42 = bool(dd <= ee) &
    a43 = bool(dd <= ff) &
    a44 = bool(dd <= gg) &
    a45 = bool(dd <= hh) &
    a51 = bool(hh <= aa) &
    a52 = bool(hh <= ee) &
    a53 = bool(hh <= ff) &
    a54 = bool(hh <= gg) &
    a55 = bool(hh <= hh) &

    b55 = a55 &
    b45 = bool(a45 = TRUE & b55 = TRUE) &
    b35 = bool(a35 = TRUE & b45 = TRUE) &
    b25 = bool(a25 = TRUE & b35 = TRUE) &
    b15 = bool(a15 = TRUE & b25 = TRUE) &
    b54 = bool(a54 = TRUE & b55 = TRUE) &
    b44 = bool(a44 = TRUE & (b54 = TRUE or b45 = TRUE)) &
    b34 = bool(a34 = TRUE & (b44 = TRUE or b35 = TRUE)) &
    b24 = bool(a24 = TRUE & (b34 = TRUE or b25 = TRUE)) &
    b14 = bool(a14 = TRUE & (b24 = TRUE or b15 = TRUE)) &
    b53 = bool(a53 = TRUE & b54 = TRUE) &
    b43 = bool(a43 = TRUE & (b53 = TRUE or b44 = TRUE)) &
    b33 = bool(a33 = TRUE & (b43 = TRUE or b34 = TRUE)) &
    b23 = bool(a23 = TRUE & (b33 = TRUE or b24 = TRUE)) &
    b13 = bool(a13 = TRUE & (b23 = TRUE or b14 = TRUE)) &
    b52 = bool(a52 = TRUE & b53 = TRUE) &
    b42 = bool(a42 = TRUE & (b52 = TRUE or b43 = TRUE)) &
    b32 = bool(a32 = TRUE & (b42 = TRUE or b33 = TRUE)) &
    b22 = bool(a22 = TRUE & (b32 = TRUE or b23 = TRUE)) &
    b12 = bool(a12 = TRUE & (b22  = TRUE or b13 = TRUE)) &
    b51 = bool(a51 = TRUE & b52 = TRUE) &
    b41 = bool(a41 = TRUE & (b51 = TRUE or b42 = TRUE)) &
    b31 = bool(a31 = TRUE & (b41 = TRUE or b32 = TRUE)) &
    b21 = bool(a21 = TRUE & (b31 = TRUE or b22 = TRUE)) &
    b11 = bool(a11 = TRUE & (b21 = TRUE or b12 = TRUE)))
 =>
    b11 = TRUE
END
```

Bibliographie

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[Cle] ClearSy System Engineering. *Atelier B*. `www.clearsy.com`.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes : A resource accounting interface for Java. In *ACM OOPSLA Conference*, 1998.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, 2000.

[Dij68] E.W. Dijkstra. Co-operating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[FGR98] Lisbeth Fajstrup, Eric Goubault, and Martin Raussen. Detecting deadlocks in concurrent systems. In *International Conference on Concurrency Theory*, pages 332–347, 1998.

[GDGF02] Antoine Galland, Damien Deville, Gilles Grimaud, and Bertil Folliot. Contrôle des ressources dans les cartes à microprocesseur. In *1er congrès Logiciel Temps Réel Embarqués (LTRE'02)*, Toulouse, France, 2002.

[Gol78] E. Mark Gold. Deadlock prediction : Easy and difficult cases. In *SIAM Journal on Computing*, volume 7, pages 320–336, 1978.

[Gou95] Eric Goubault. *The Geometry of Concurrency*. PhD thesis, École Polytechnique, 1995.

[Gou00] Eric Goubault. Geometry and concurrency : A user's guide. *Mathematical Structures in Computer Science*, 10(4) :411–425, 2000.

[Gri00] Gilles Grimaud. *CAMILLE : un système d'exploitation ouvert pour carte à microprocesseur*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, 2000.

[Hab69] A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7) :373–377, 1969.

[INR] INRIA, projet Cristal. *The Ocaml language*. `www.ocaml.org`.

[LR01] M. Lawley and S. Reveliotis. Deadlock avoidance for sequential resource allocation systems : Hard and easy cases. *The International Journal of FMS*, 13(4) :385–404, 2001.

[Mao02] Maosco Ltd. *Multos website*, 2002. `www.multos.com`.

[RF96] S. Reveliotis and P. Ferreira. Deadlock avoidance policies for automated manufacturing cells. In *IEEE Trans. on Robotics & Automation*, volume 12, pages 845–857, 1996.

[Sun02] Sun Microsystems. *The Javacard 2.2 specifications*, 2002. `http://java.sun.com/products/javacard/`.

[WN94] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994.

Index

$[\![p]\!]^0$, 12
$[\![p]\!]^1$, 13
$[\![p]\!]^2$, 24
$[\![p]\!]^3$, 25
$[\![p]\!]^{0'}$, 12
$[\![p]\!]^{01}$, 25
$[\![p]\!]^{02}$, 25
état, 30
Σ_n, 14
$\approx^\circ$, 8
ϵ, 7, 11
$\leq^2$, 17
$\|$, 7, 11
$\preceq^0$, 14
$\preceq$, 15
$\rightarrow$, 9
$\sqsupseteq$, 17
$\sqsubseteq$, 17
$\xrightarrow[\neq]{l}$, 9
$\xrightarrow[\neq]{}$, 9
$\xrightarrow{(c,\delta)}$, 11
$\xrightarrow{\epsilon}$, 7, 11
$\xrightarrow{fusion}$, 12, 32
$\xrightarrow{l}$, 9
$\xrightarrow{x}$, 7
$lth(\_)$, 31
$merge_{\preceq}$, 16
$sort_{\preceq}$, 16
$v_1 \oplus v_2$, 31

algorithme du banquier, 5
allocations bloquantes, 4

bisimilarité forte, 8
bisimulation forte, 8
bloc, 11
borne inférieure, 36
borne supérieure, 36

choix non-déterministe, 40
coût, 10, 30, 31
connexion de Galois, 27
contexte, 8
contexte d'évaluation, 7
contrôle dynamique, 3

deadlock, 4
détection & résolution, 4

état dangereux, 5
état sûr, 5
états dangereux, 4
évitement, 4
exécution concurrente, 7, 11

famine, 4, 41
fusion, 12

graphes d'avancement, 5

interblocage, 4

langage de processus, 7
liste normalisée, 20
liste de variations, 9

ordonnancement, 9

prévention, 4
processus vide, 7, 11
progress graph, 5

répétition finie, 41
résumé, 1
règles d'évaluation, 7, 11
règles structurelles, 7, 11
regroupement de blocs, 12

séquence, 7, 11
simplifiabilité, 20
simulation, 8
simulation forte, 8

systèmes d'allocation de ressource, 6

trace d'exécution, 9
transitions étiquetées, 7

variation, 7
variation totale, 10
vecteur d'états, 30
vecteurs candidats, 37

zones interdites, 5