Composition parallèle pour MSPML sémantique et implantation

Radia Benheddi

Stage de Master

sous la direction de
Frédéric Loulergue
Laboratoire d'Algorithmique, Complexité et Logique
61, avenue du général de Gaulle
94010 Créteil cedex – France
loulergue@univ-paris12.fr

Avril-Septembre 2005

Remerciements

Je voudrais remercier Frédéric Loulergue pour m'avoir donner l'opportunité de participer au projet PROPAC, et de m'avoir encadré et guidé au long de mon stage.

Je remercie mes parents, mes soeurs et mon frère, en particulier mon père de m'avoir tous le temps soutenu et encouragé dans mes études.

Je remercie Nabil pour son soutien et ses conseils.

Résumé

De nos jours, les machines parallèles seules sont aptes à délivrer les puissances de calcul importantes.

Le parallélisme de données est un paradigme de programmation parallèle dans lequel un programme décrit une séquence d'actions sur des tableaux à accès parallèle. Le modèle Bulk Synchronous Parallel ou BSP vise à maximiser la portabilité des performances en ajoutant une notion de processus explicites au parallélisme de données. Un programme BSP est écrit en fonction du nombre de processeurs de l'architecture sur laquelle il s'exécute. Le modèle d'exécution BSP sépare synchronisation et communication et oblige les deux à être des opérations collectives. Il propose un modèle de coût fiable et simple permettant de prévoir les performances de façon réaliste et portable.

BSML est un langage fonctionnel pour les programmes parallèles BSP. Il permet la programmation data-parallèle basée sur une structure de données parallèle et polymorphe qui est manipulée à l'aide d'opérations dédiées. En particulier, l'ordre de lecture et d'exécution BSP sont identiques. Les difficultés de la programmation SPMD sont donc éliminées. Ainsi, les inter-blocages sont impossibles et le déterminisme est garanti.

Minimally synchronous Parallel ML (MSPML) est un langage fonctionnel parallèle qui a la même sémantique de haut niveau que BSML mais une sémantique de bas niveau et une implantation complètement différentes. MSPML possède ainsi, une sémantique asynchrone (c'est à dire sans barrières de synchronisation globale). Cette propriété s'avère plus convenable pour des programmes non équilibrés.

La composition parallèle spatiale est une opération permettant d'évaluer deux programmes parallèles sur deux parties différentes d'une même machine. Dans la sémantique de BSML, l'opération de composition parallèle telle qu'elle est définie, nécessite que deux expressions composées parallèlement s'évaluent en utilisant le même nombre de barrières de synchronisation. Une contrainte qui, bien évidement, n'est pas souhaitable. MSPML résout ce problème en offrant une sémantique d'évaluation asynchrone, ce qui rend l'opération de composition parallèle plus efficace puisqu'un de ses avantages est de pouvoir adapter les programmes aux spécificités des machines hétérogènes.

Dans ce travail, on ajoute l'opération de composition parallèle à MSPML en concevant une nouvelle sémantique à petit pas (distribuée) et une nouvelle sémantique à grand pas qui étendent les sémantiques précédentes, en prouvant leurs

confluences et enfin en ajoutant cette opération à l'implantation de MSPML.

Mots clés Conception de langage de programmation parallèle, Sémantique formelle, composition parallèle, confluence, Typage.

Abstract

Nowadays, only the parallel machines are ready to deliver the important computing powers.

The parallelism of data is a paradigm of parallel programming in which a program describes a sequence of actions on vectors with parallel access. The model Bulk Synchronous Parallel (BSP) aims at maximizing the portability of the performances by adding a concept of explicit processes to parallelism of data. A BSP program is written according to the number of processors of the architecture on which it is carried out. The BSP execution model separates synchronization and communication and obliges both to be collective operations. It proposes a reliable and simple cost model making it possible to predict the performances in a realistic and portable way.

BSML is a functional language for BSP parallel programs. It allows the data-parallel programming based on a parallel and polymorphic data structure which is handled using dedicated operations. In particular, the reading order and execution BSP are identical. The difficulties of SPMD programming are eliminated. Thus, deadlocks are impossible and the determinisme is guaranteed.

Minimally synchronous Parallel ML (MSPML) is a parallel functional language which has same high level semantics as BSML but a completely different low level semantics and an implementation. MSPML has an asynchronous semantics (i.e. without barriers of global synchronization). This property proves more suitable for unbalanced programs.

The parallel composition is an operation making it possible to evaluate two parallel programs out of two different parts from the same machine. In BSML semantics, the operation of parallel composition such as it is defined, requires that two expressions compound in parallel are evaluated by using the same number of synchronization barriers. A constraint which is not desirable. MSPML solves this problem by offering an asynchronous semantics of evaluation which makes the operation of parallel composition more efficient since one of its advantages is to be able to adapt the programs to specificities of the heterogeneous machines.

In this work, we add the operation of parallel composition to MSPML by designing a new low level semantics and a new high level semantics which extends the preceding semantics and proving their confluences. Finally, we add this operation to the MSPML implementation.

Key words Design of functional programming language,formal Semantics, parallel composition, confluence, Typing.

Table des matières

Chapitre 1

Introduction

De nos jours, les machines parallèles apparaissent incontournables, elles seules sont aptes à délivrer les puissances de calcul importantes dont ont besoin un nombre toujours croissant d'applications, comme la simulation de phénomènes complexes (naturels, physiques, chimiques,etc.) ou encore la gestion de bases de données parallèles.

La programmation de systèmes parallèles connaît deux solutions possibles dont chacune occupe une extrémité :

La programmation séquentielle qui est utilisée via la parallélisation automatique réalisée par le compilateur. Bien qu'idéale pour sa facilité d'écriture, elle est surtout limitée à l'utilisation de tableaux et de boucles [12]. Cette approche oblige le compilateur à découvrir le parallélisme implicite, et à générer le meilleur programme parallèle possible. Le compilateur parallélisant doit dans le pire des cas produire un bon algorithme parallèle à partir d'une "spécification" séquentielle. Or, ce n'est pas toujours faisable, d'où la nécessité de mettre au point des méthodes de programmation parallèle plus générales mais aussi simples d'utilisation que possible.

La programmation concurrente est souvent utilisée pour écrire des algorithmes parallèles en combinant un langage séquentiel avec une bibliothèque de communication tel que MPI [26]. Cette approche permet de définir l'algorithme parallèle ainsi que les détails de sa réalisation par des protocoles de communication. Cependant, la difficulté de mise au point de tels programmes est grande du fait de l'indéterminisme et de la possibilité de blocage, ce qui est confirmé par la très haute complexité des problèmes de validation associés [1]. Comme le sens d'un programme concurrent est en général très complexe, son temps de calcul est difficile à établir, ce qui peut gêner la portabilité des performances. De plus, les programmes écrits avec une bibliothèque comme MPI sont dans le style SPMD (Single Program Multiple Data). Cela veut dire que le même programme est écrit pour tous les processeurs de la machine. Toutefois, celui-ci contient des expressions dépendantes du numéro du processeur qui est lié extérieurement. Ceci amène plusieurs défauts, d'une part, il peut être

difficile pour le programmeur - et il est indispensable - de distinguer quelles parties du programme sont locales, c'est-à-dire, dépendantes du processeur sur lequel elles s'exécutent de celles qui sont globales, c'est-à-dire, indépendantes du processeur et qui décrivent le comportement global de l'algorithme parallèle. D'autre part, l'ordre de lecture du programme peut ne plus correspondre à l'ordre d'exécution ce qui rend malaisée la mise au point.

D'où on trouve une voie intermédiaire dans laquelle un programme décrit une séquence d'actions sur des tableaux à accès parallèle [8]. Le modèle de programmation est synchrone et outre qu'il y ait une plus grande facilité de conception des programmes que dans le cas concurrent, le non-blocage est garanti. De plus, les possibilités d'indéterminisme sont assez réduites pour que des systèmes de preuves de programmes [9] aient une complexité proche de celle des programmes séquentiels. Il est prévu que l'exécution d'un programme data-parallèle soit désynchronisée par le compilateur ou le système dynamique en vue de maximiser ses performances.

Bulk Synchronous Parallel (BSP) est un modèle introduit par Valiant [27, 25] pour offrir un degré de portabilité et de performances maximales en ajoutant une notion de processus explicites au parallélisme de données.

Les principaux avantages du modèle BSP sont :

- l'absence de blocage ainsi que l'évitement ou la restriction maximale de l'indéterminisme [20] dus à la séparation entre la synchronisation et la communication et oblige ces deux dernières à être des opérations collectives ;
- maximise la portabilité des performances [28] en ajoutant une notion de processus explicites au parallélisme de données.

La théorie de la preuve des programmes BSP [18] est elle aussi proche en complexité du cas séquentiel. Le modèle BSP a été utilisé avec succès pour une large variété de problèmes : le calcul scientifique [6, 5], les algorithmes génétiques [11], la programmation génétique [15], les réseaux de neurones [24], les bases de données parallèles [3, 2], les solveurs de contraintes [17], etc.

Le langage Bulk Synchronous Parallel ML (BSML) est une bibliothèque développée pour Objective Caml [29, 13]. Elle permet la programmation data-parallèle basée sur une structure de données parallèles polymorphe. Les programmes sont des fonctions (séquentielles) que l'on peut programmer en Caml mais manipulent cette structure de données parallèle à l'aide d'opérations dédiées. En particulier, l'ordre de lecture et d'exécution BSP sont identiques. Les difficultés de la programmation SPMD sont donc supprimées. Puisque BSML suit le modèle d'exécution BSP, les inter-blocages sont impossibles et le déterminisme est garanti.

Le mécanisme de synchronisation globale imposé par le modèle BSP soulève quelques problèmes de surcoût de la barrière de synchronisation globale où par exemple la synchronisation d'un processeur à une étape donnée alors qu'il n'est engagé dans aucune communication. De plus, dans la sémantique actuelle de BSML, l'opération de composition parallèle (c'est une opération permettant d'évaluer deux programmes parallèles sur deux parties différentes d'une même machine) telle qu'elle est définie, nécessite que deux expressions composées parallèlement s'évaluent en uti-

lisant le même nombre de barrières de synchronisation. Cette nécessité disparaît dans une sémantique d'évaluation asynchrone.

Minimally synchronous Parallel ML (MSPML) est un langage fonctionnel parallèle qui a la même sémantique de haut niveau que BSML mais une sémantique de bas niveau et une implantation complètement différentes. MSPML possède ainsi, une sémantique asynchrone (c'est à dire sans barrières de synchronisation globale). Cette propriété s'avère plus convenable pour des programmes non équilibrés. MSPML résout le problème de la composition parallèle en offrant une sémantique d'évaluation asynchrone, ce qui rend l'opération de composition parallèle plus efficace puisqu'un de ses avantages est de pouvoir adapter les programmes aux spécificités des machines hétérogènes.

Dans des travaux futurs, MSPML et BSML vont être mixés pour obtenir un nouveau langage parallèle fonctionnel nommé Departemental Meta-computing ML (DMML) pour le *metacomputing*, c'est-à-dire, pour la programmation de grappes de machines parallèles (qui sont elles-mêmes souvent des grappes de PC) que nous appelons méta-ordinateurs. Plusieurs programmes BSML s'exécutent sur chaque noeud parallèle et sont coordonnés par un programme MSPML [16].

Ce manuscrit est organisé comme suit :

Premièrement, on donne une description informelle du langage MSPML dans le chapitre 2 écrit en collaboration avec mon collègue Abdeltouab Belbekkouche qui travaille sur *le mécanisme de communication et la tolérance aux pannes dans MSPML*. Ensuite, on présente formellement la nouvelle sémantique à grands pas de MSPML dans le chapitre 3. La sémantique distribuée est présentée dans le chapitre 4. On présente ensuite, au chapitre 5, les éléments de l'implantation. Dans le chapitre 6 on présente les preuves de confluence de la sémantique à petits pas et à grands pas ainsi que des propositions pour la sûreté de typage. Enfin, dans le chapitre 7, on donne nos conclusions à propos des travaux menés et nos perspectives pour les travaux futurs.

Chapitre 2

MSPML : Description informelle

Minimally Synchronous Parallel ML (MSPML) est un langage fonctionnel (extension d'Objective Caml).

Un programme MSPML peut être vu comme un programme séquentiel usuel qui manipule une structure de données parallèles (appelée vecteur parallèle). Cela est différent de la programmation SPMD (*Single Program Multiple Data*) où le programmeur utilise un langage séquentiel avec une bibliothèque de communication (comme MPI [26]).

Un programme parallèle SPMD est donc de multiples copies d'un programme séquentiel, qui échangent des messages en utilisant une bibliothèque de communication. Dans ce cas, les messages et les processus sont explicites, mais les programmes peuvent être non déterministes ou encore contiennent des inter-blocages.

Un autre inconvénient de la programmation SPMD est l'utilisation d'une variable contenant le nom du processus (usuellement appelé "pid" pour *Process Identifier*) qui ne dépend pas du programme source.

Dans ce chapitre, on présente le modèle de coûts et d'exécution pour MSPML, puis on présente son noyau et son architecture. Enfin, on donne une comparaison entre MSPML et BSML.

2.1 Modèle de coûts et d'exécution

Plutôt que de passer directement à la conception d'un modèle et d'un langage à deux niveaux pour l'utilisation de grappes de machines parallèles, les créateurs de MSPML ont d'abord considéré la conception d'un langage proche de BSML mais sans les barrières de synchronisation.

Il est souvent admis que les barrières de synchronisation ne sont pas un handicap pour les performances (dans le cas *d'une seule* machine parallèle), notamment parce qu'une vue globale du calcul permet des optimisations qui ne sont pas possibles dans le cas d'un parallélisme moins structuré (voir par exemple [19]) ou que l'éventualité de performances un peu moins bonnes n'est pas un désavantage suffisant par rapport

à la plus grande facilité de conception et de correction/vérification des algorithmes et programmes.

Toutefois il y a de nombreux programmes parallèles implémentés, en particulier en MPI, qui ne suivent pas le modèle BSP mais pour lesquels on souhaite pouvoir raisonner sur le coût. C'est ce qui a conduit au modèle *BSP without barrier* [23] (BSPWB) puis au modèle MPM [22, 7].

BSPWB est un modèle directement inspiré du modèle BSP. Il propose de remplacer la notion de super-étape par la notion de m-étape définit comme suit. À chaque m-étape, chaque processeur effectue une phase de calcul suivie par une phase de communication. Durant la phase de communication, les processeurs échangent les données dont ils ont besoin pour la m-étape suivante.

La machine parallèle est caractérisée par les trois paramètres suivants (les deux derniers sont exprimés comme multiples de la puissance de calcul des processeurs) :

- le nombre de processeurs p,
- la latence L du réseau,
- le temps g pour échanger un mot entre deux processeurs.

Le temps nécessaire à un processeur i pour exécuter une m-étape s est $t_{s,i}$ borné par T_s le temps nécessaire à l'exécution de la m-étape s par la machine parallèle.

T_s est défini inductivement par :

$$\begin{cases} T_1 = max\{w_{1,i}\} + \max\{g \times h_{1,i} + L\} \\ T_s = T_{s-1} + max\{w_{s,i}\} + max\{g \times h_{s,i} + L\} \end{cases}$$

où $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$ où R est le nombre de m-étapes du programme et $w_{s,i}$ et $h_{s,i}$ sont respectivement le temps de calcul local au processeur i durant la m-étape s et $h_{s,i} = max\{h^+_{s,i}, h^-_{s,i}\}$ où $h^+_{s,i}$ (resp. $h^-_{s,i}$) est le nombre de mots reçus (resp. envoyés) par le processeur i durant la m-étape s.

Dans ce modèle il y a toutefois une barrière implicite à chaque étape, le coût de la barrière elle-même étant nul. De ce fait ce modèle est une approximation trop grossière. Une meilleure borne $\Phi_{s,i}$ est donnée par le modèle *Message Passing Machine* [22]. Les paramètres de ce modèle sont identiques à ceux du modèle BSPWB.

On utilise l'ensemble $\Omega_{s,i}$ pour un processeur i et une m-étape s (Figure 2.1) défini par :

$$\Omega_{s,i} \ = \ \{j | \text{processeur } j \text{ envoie un message au processeur } i \text{ à la m-étape } s\} \bigcup \{i\}$$

Les processeurs de l'ensemble $\Omega_{s,i}$ sont appelés "partenaires entrants" du processeur i à la m-étape s. La borne $\Phi_{s,i}$ est définie inductivement par :

$$\begin{cases} \Phi_{1,i} = \max\{w_{1,j} | j \in \Omega_{1,i}\} + (g \times h_{1,i} + L) \\ \Phi_{s,i} = \max\{\Phi_{s-1,j} + w_{s-1,j} | j \in \Omega_{s,i}\} + (g \times h_{s,i} + L) \end{cases}$$

où $h_{s,i} = \max\{h^+_{s,i}, h^-_{s,i}\}$ pour $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$.

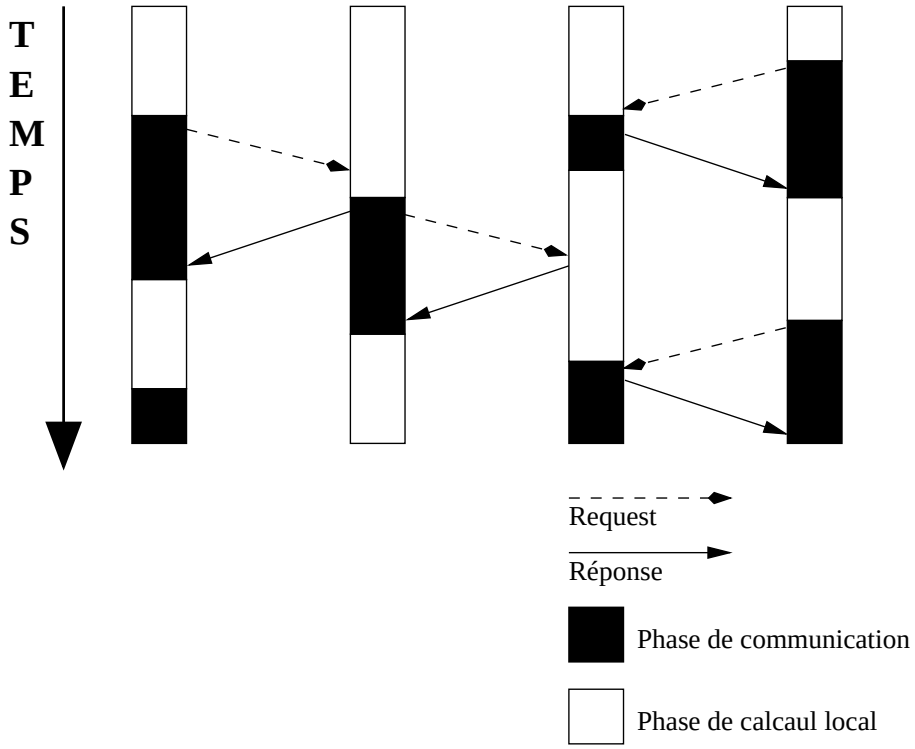

FIG. 2.1 – Le modèle MPM

Le temps d'exécution pour un programme est donc borné par :

$$\Psi = \max\{\Phi_{R,j} | j \in \{0, 1, \ldots, p-1\}\}$$

Le modèle MPM prend en compte le fait qu'un processeur ne se synchronise qu'avec chacun de ses partenaires entrants et est donc plus précis que BSPWB. Les expériences menées montrent que ce modèle s'applique bien à MSPML[21].

2.2 Le noyau de la bibliothèque

La bibliothèque MSPML est basée sur les primitives données dans la figure 2.2. Ils donnent l'accès aux paramètres du modèle Message Passing Machine (MPM) utilisé. En particulier, la fonction **p()** retourne le nombre statique de processeurs de la machine parallèle. Cette valeur ne change pas durant l'exécution. Il y a aussi un type abstrait polymorphique α **par** qui représente le type des vecteurs parallèles de longueur p d'objets de type α, un seul objet par processeur. La non-imbrication des types **par** est assurée par le système de typage [20].

Les constructeurs parallèles opèrent sur les vecteurs parallèles. Ces vecteurs parallèles sont crées par la primitive **mkpar**, ainsi, (**mkpar** f) stocke (fi) dans le processeur i pour i entre 0 et $(p-1)$. On écrit habituellement **fun** $pid \rightarrow e$ pour f

p	:	$unit- > int$
g	:	$unit- > float$
l	:	$unit- > float$
mkpar	:	$(int- >' a)- >' a\ par$
apply	:	$('a- >' b)\ par- >' a\ par- >' b\ par$
get	:	$'apar- > int\ par- >' a\ par$
mget	:	$(int- >' a)\ par- > (int- > bool)\ par- > (int- >' aoption)\ par$
at	:	$'apar- > int- >' a$
juxta	:	$int- > (unit- >' a\ par)- > (unit- >' a\ par)- >' a\ par$

FIG. 2.2 – Le noyau de la bibliothèque MSPML.

afin de montrer que l'expression e peut être différente sur chaque processeur. Cette expression est appelée locale : elle est à l'intérieur de la fonction **mkpar** et sa valeur dépend du processeur local sur lequel elle se trouve. L'expression (**mkpar** f) est un objet parallèle global. Par exemple l'expression **mkpar**(**fun** $pid\ \rightarrow\ pid$) sera évaluée en un vecteur parallèle $\langle\ 0\ ,\ldots,\ p-1\ \rangle$.

Dans le modèle MPM, un algorithme est écrit comme étant une combinaison entre des calculs locaux asynchrones et des phases de communication. Les phases asynchrones sont programmées avec **mkpar** et **apply**. Par exemple, l'expression **apply** (**mkpar** f) (**mkpar** e) stocke (fi) (ei) dans le processeur i.

Les phases de communication sont réalisées par **get** et **mget**. La sémantique du **get** est donnée par :

$$\textbf{get}\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\langle\ i_0\ ,\ldots,\ i_{p-1}\ \rangle = \langle\ v_{i0\%p}\ ,\ldots,\ v_{i0\%p-1}\ \rangle$$

où % est le modulo.

La fonction **mget** est une généralisation qui permet d'avoir les données à partir de différents processeurs durant la même m-étape et de délivrer différents messages à de différents processeurs.

Dans le type de **mget** dans la figure 2.2, α option est définie comme suit :

$$\textbf{type}\ \alpha\ \text{option} = \text{None} \mid \text{Some}\ \textbf{of}\ \alpha.$$

La sémantique de la fonction **mget** est :

$$\textbf{mget}\langle\ f_0\ ,\ldots,\ f_{p-1}\ \rangle\langle\ b_0\ ,\ldots,\ b_{p-1}\ \rangle = \langle\ g_0\ ,\ldots,\ g_{p-1}\ \rangle$$

où $g_i = \textbf{fun}\ j\ \rightarrow\ \textbf{if}\ b_i\ j\ \textbf{then}\ \text{Some}\ (f_j\ i)\ \textbf{else}\ \text{None}$

Le langage complet contient aussi une conditionnelle synchrone :

$$\textbf{if}\, e\, \textbf{at}\, n\, \textbf{then}\, e_1\, \textbf{else}\, e_2$$

Selon la valeur du vecteur parallèle au processeur donné par la valeur n l'expression est évaluée en v_1 (la valeur obtenue par l'évaluation de e_1)ou en v_2 (la valeur

obtenue par l'évaluation de e_2). Mais, Objective Caml est un langage stricte, ainsi, cette opération conditionnelle synchrone ne peut être définie comme une fonction. Pour cela, la bibliothèque MSPML contient la fonction **at** pour être utilisée dans les constructions suivantes :

- **if at** e n **then** ... **else** ...
- **match**(**at** e n) **with**...

at exprime la phase de communication. La conditionnelle globale est nécessaire pour exprimer des algorithmes tels que :
Repeat
Itération Parallèle
Until erreur locale maximale $< \epsilon$

Sans le **at**, le contrôle globale ne peut prendre en compte les données calculées localement.

2.3 Exemples

On présente maintenant quelques exemples qui font partie de la bibliothèque MSPML.

2.3.1 Les fonctions les plus utilisées

Quelques fonctions usuelles peuvent être définie en utilisant que les primitives. Par exemple la fonction **replicate** crée des vecteurs parallèles qui contiennent la même valeur partout. La primitive **apply** peut être utilisée rien que pour les vecteurs parallèles de fonctions qui prennent un seul argument. Pour les fonctions à deux arguments on a besoin de définir la fonction **apply2**.

let replicate x = **mkpar**(**fun** pid → x)
let apply2 f v1 v2 = **apply**(**apply** f v1) v2

Il est aussi très commode d'appliquer la même fonction séquentielle à chaque processeur. Cela peut être fait en utilisant les fonctions **parfun** : elles différent seulement dans le nombre d'arguments :

let parfun f v = **apply** (replicate f)v
let parfun2 f v1 v2 = **apply**(parfun f v1) v2
let parfun3 f v1 v2 v3 = **apply**(parfun2 f v1 v2) v3

(applyat n f_1 f_2 v) applique la fonction f_1 au processeur n et la fonction f_2 aux autres processeurs :

let applyat n f1 f2 v = **apply**(**mkpar**(**fun** i → (**if** i = n **then** f1 **else** f2))) v

parpair_of_pairpar transforme un vecteur parallèle de paires en une paire de vecteurs parallèles :

let fst (x, y) =x
let snd(x, y) = y
let parpair_of_pairpar vv = (parfun fst vv, parfun snd vv)

2.3.2 Les fonctions de communication

La sémantique de la fonction d'échange totale est donnée par :
$totex\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle = \langle\ f\ ,\ldots,\ f\ ,\ldots,\ f\ \rangle$
où $\forall i.(0 \leq i < p-1) \Rightarrow\ (f\ i) = v_i$. Le code est comme présenté ci-dessous où, **noSome** enlève le constructeur **Some** et **compose** est la fonction de composition :

(∗ val totex: α par → (int → α) par ∗)
let totex vv = (parfun compose noSome)
(mget (parfun(**fun** v i → v)vv)(replicate(**fun** i → **true**)))

Son coût parallèle est $(p-1) \times s \times g + L$, où s dénote la taille en mots de la plus grande valeur v qui se trouve sur un certain processeur n. À partir de la fonction d'échange total, on peut obtenir une version qui retourne un vecteur parallèle de listes :

(∗ val totex_list: α par → α list par ∗)
let totex_list v = (parfun2 List.map(totex v)(replicate(procs()))

où :

$$\begin{cases} \textit{(∗val List.map: (} \alpha \rightarrow \beta \textit{) → } \alpha \textit{ list → } \beta \textit{ list ∗)} \\ \text{List.map } f\ [v_0;\ldots;v_n] = [(f\ v_0);\ldots;(f\ v_n)] \\ \text{procs() } = [0;\ldots;\mathbf{p}()\text{-}1]. \end{cases}$$

La sémantique de la diffusion est :

$$\text{bcast}\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ r\ \ =\ \ \langle\ v_{r\%p}\ ,\ldots,\ v_{r\%p}\ \rangle$$

La fonction broadcast_direct qui réalise la diffusion peut être écrite comme suit :

(∗ bcast_direct: int → α par → α par ∗)
let bcast_direct root vv = **get** vv (replicate root)

Son coût parallèle est $(p-1) \times s \times g\ +\ L$, où s dénote la taille en mots de la valeur v_n qui se trouve sur le processeur n.

La bibliothèque standard de MSPML contient une collection de ces fonctions qui facilitent l'écriture de programmes. Ainsi, il est exactement similaire d'écrire des programmes MSPML ou d'écrire des programmes en utilisant des patrons data-parallèles, mais, avec MSPML il est possible d'écrire ses propres patrons comme étant des fonctions de haut niveau si la bibliothèque standard ne fournit pas les fonctions nécessaires.

Quelques fonctions de la bibliothèque standard sont récursives. Par exemple, il existe une fonction broadcast qui est évaluée en $\log p$ m-étapes au lieu d'une m-étape :

```
let bcast_logp root vv =
  let from n =
    mkpar(fun i → let j=natmod (i+(p())−root) (p()) in
      if (n/2<=j)&&(j<n) then i−(n/2) else i) in
  let rec aux n vv =
    if n<1 then vv else get (aux (n/2) vv) (from n)
  in aux (p()) vv
```

2.4 Le mécanisme de communication

La désynchronisation de notre langage MSPML, nous a conduit à penser à avoir un moyen de stockage des valeurs de chaque super-étape dans le cas où un processeur pourrait en avoir besoin ultérieurement. c'est pour cela qu'un environnement de communication a été introduit.

Durant l'exécution d'un programme MSPML, pour chaque processus i, le système possède une variable $mstep_i$ contenant le nombre du m-step actuel. À chaque fois une expression **(get vv vi)**, est évaluée à un processus i donné :

1. $mstep_i$ est incrémentée par un.
2. Le couple contenant : la valeur que tient ce processus dans le vecteur parallèle vv et la valeur de $mstep_i$ est sauvegardée dans l'environnement de communication. Un environnement de communication peut être vu comme une liste d'association qui relie les nombres de m-étape avec les valeurs des processus à ces m-étapes.
3. La valeur j que tient ce processus dans le vecteur parallèle vi désigne le numéro de processus duquel le processus i veut recevoir une valeur. Ainsi, le processus i envoie une requête au processus j : il demande la valeur à la m-étape $mstep_i$. Quand le processus j reçoit la requête (des *threads* sont dédiés au traitement de telles requêtes, donc, le travail du processus j n'est pas interrompu), deux cas se présentent :
   - $mstep_j \geq mstep_i$: cela veut dire que le processus j a déjà atteint la même m-étape que le processus i. Ainsi, le processus j accède dans son environnement de communication à la valeur associée à la m-étape $mstep_i$ et l'envoie au processus i.
   - $mstep_j < mstep_i$: rien ne peut se faire jusqu'à ce que le processus j atteigne la même m-étape que le processus i.

   Si $i = j$, l'étape 3 n'est pas exécutée.

2.5 Comparaison avec BSML

Bulk Synchronous Parallel ML (BSML) a les mêmes opérations que MSPML, Mais avec une seule différence. Les communications (suivies immédiatement par une

barrière de synchronisation) sont réalisées par la primitive **put**. Son type est le suivant :

$$\textbf{put}\text{:(int} \rightarrow \alpha \text{ option) } \textbf{par} \rightarrow \text{(int} \rightarrow \alpha \text{ option) } \textbf{par}$$

Considérons l'expression suivante :

$$\textbf{put } (\textbf{mkpar } (\textbf{fun i} \rightarrow \text{fs}_i)) \tag{2.1}$$

Pour envoyer la valeur v du processeur j au processeur i, la fonction fs_j au processeur j doit être comme étant (fs_j i) évaluée en (Some v). Pour n'envoyer aucune valeur à partir du processeur j au processeur i, (fs_j i) doit être évaluée en None.

L'expression (2.1) s'évalue en un vecteur parallèle contenant une fonction fd_i des messages délivrés au niveau de chaque processeur. au processeur i, (fd_i j) s'évalue en None si le processeur j n'envoie aucun message au processeur i ou s'évalue en (Some v) si le processeur j envoie la valeur v au processeur i.

Cette primitive peut être utilisée pour programmer les fonctions **get** et **mget**.Mais ces deux fonctions sont moins efficaces que les primitives de MSPML : elles nécessitent deux super-étapes BSP, par conséquent deux barrières de synchronisation.

Donc pour le moment la principale différence entre la programmation BSML et la programmation MSPML est que les communications sont faites avec le *style get* dans MSPML et le *style put* dans BSML.

Par exemple, le broadcast cité ci-dessus peut être écrit en BSML :

```
let bcast_direct root vv =
  let mkmsg = mkpar(fun i v dst → if i=root
  then Some v else None) in parfun noSome
    (apply (put (apply mkmsg vv)) (replicate root))
```

Les programmes BSML et MSPML qui utilisent rien que la fonction bcast_direct pour la communication doivent être identiques, mais leurs coûts vont être différents. Par conséquent, il est facile de réécrire les programmes BSML pour obtenir les programmes MSPML. La seule difficulté est quand le programme BSML utilise directement la primitive **put**, à partir des fonctions de la bibliothèque standard : quelques réécritures compliquées sont nécessaires.

Chapitre 3

MSPML : Sémantique à grands pas

Ce chapitre traite la sémantique formelle à grand pas de MSPML. C'est la sémantique qui correspond au modèle de programmation. Elle est très semblable à la sémantique de BSML [21] (mais l'opérateur **get** est ici une primitive alors que c'est possible de la définir en utilisant la primitive **put**). Pour simplifier on se limite aux communications **get** et **ifat**.

3.1 MSPML sans juxtaposition

3.1.1 Syntaxe et typage

Le noyau que nous considérons ici est donné par la grammaire donnée à la figure 3.1.

Dans cette syntaxe nous mettons en oeuvre un typage simple en expressions locales et globales, en typant les variables. Une variable est soit locale, $x : \mathcal{L}$, soit globale, $x : \mathcal{G}$.

Pour notre langage fonctionnel les expressions du langage sont essentiellement les termes λ-calcul avec constantes et construction de liaison **let**.

L'ensemble des opérateurs *op* contient les opérations arithmétiques et opérateur de point fixe[1].[14]. **mkpar**, **apply**, **get** et **ifat** sont des opérateurs parallèles dont **get** et **ifat** sont de communication. **fst** et **snd** sont des opérateurs de pairs.

La syntaxe de la figure 3.1 est la syntaxe du programmeur, l'évaluation pouvant produire des vecteurs parallèles :

$$e ::= ... \mid \langle\, e\,, \ldots,\, e\,, \ldots,\, e\,\rangle$$

[1]L'opérateur **fix** remplace la construction **let rec** qui n'est pas pleinement satisfaisante, car dans les évaluations **let rec** ne peut être remplacer par un texte qui lui même contient un nom d'une fonction

e'	$::= \quad x$	(variables)
	$\mid c$	(constantes)
	$\mid$ **fun** $x : \tau \ \rightarrow \ e'$	(abstraction)
	$\mid$ **let** $x : \tau = e'$ **in** e'	(liaison)
	$\mid op$	(opérateurs)
	$\mid (e' \ e')$	(application)
	$\mid (e', e')$	(paires)
	$\mid$ **if** e' **then** e' **else** e'	(conditionnelle)
	$\mid$ **fst** e'	(premier de la paire)
	$\mid$ **snd** e'	(deuxième de la paire)
	$\mid$ **mkpar** e'	(vecteur parallèle)
	$\mid$ **apply** $e' \ e'$	(application parallèle)
	$\mid$ **get** $e' \, e'$	(communication)
	$\mid$ **if** e' **at** e' **then** e' **else** e'	(conditionnelles globale)

FIG. 3.1 – La syntaxe du noyau du langage

Typage

Dans la syntaxe nous ne distinguons pas expressions locales et globales comme dans le BSλ-calcul mais simplement les variables. Le typage des expressions est établit sont établit par le système de typage présenté dans les figures 3.2 et 3.3. Ce système de typage contient des jugements de la forme $E \vdash e : \tau$ qui veut dire "Dans l'environnement de typage E, l'expression e est de type τ". L'environnement E donne que les types des variables libres dans e. $\{x : \tau\}$ veut dire que la variable x est du type τ dans cet environnement. Pour $\emptyset \vdash e : \tau$ on écrit $e : \tau$.

Pour éviter l'emboîtement, l'ordre suivant est appliqué :

$$\mathcal{L} < \mathcal{L} \qquad \mathcal{L} < \mathcal{G} \qquad \mathcal{G} < \mathcal{G}$$

C'est à dire qu'il est impossible d'obtenir une valeur locale à partir d'une valeur globale. Par exemple, la règle (3.5) force le type du résultat de la fonction, Pour que τ_2 soit plus grand que le type τ_1 de l'entrée : par conséquent, il est impossible d'avoir un argument global avec un résultat local.

3.1.2 Règles d'évaluation

Nous présentons là une sémantique à grand pas, associant expressions et valeurs. Les valeurs sont définies par la grammaire suivante :

$$\frac{}{E \vdash x : E(x)} \tag{3.1}$$

$$\frac{}{E \vdash c : \mathcal{L}} \tag{3.2}$$

$$\frac{}{E \vdash op : \mathcal{L}} \tag{3.3}$$

$$\frac{E \vdash e : \tau}{E \vdash (\mathbf{fix}\, e) : \tau} \tag{3.4}$$

$$\frac{E + \{x : \tau_1\} \vdash e : \tau_2 \quad \text{if } \tau_1 < \tau_2}{E \vdash (\mathbf{fun}\ x{:}\tau_1 \rightarrow e) : \tau_2} \tag{3.5}$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{if } \tau_2 < \tau_1}{E \vdash (e_1\ e_2) : \tau_1} \tag{3.6}$$

$$\frac{E \vdash e_1 : \tau_2 \qquad E + \{x : \tau_1\} \vdash e_2 : \tau_3 \quad \text{if } \tau_1 = \tau_2 \text{ and } \tau_2 < \tau_3}{E \vdash \mathbf{let}\ x{:}\tau_1 = e_1\ \mathbf{in}\ e_2 : \tau_3} \tag{3.7}$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{with } \tau_3 = \tau_2 \text{ if } \tau_1 < \tau_2 \text{ else } \tau_3 = \tau_2}{E \vdash (e_1, e_2) : \tau_3} \tag{3.8}$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{if } \tau_2 < \tau_1}{E \vdash \mathbf{fst}\ (e_1, e_2) : \tau_1} \tag{3.9}$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{if } \tau_1 < \tau_2}{E \vdash \mathbf{snd}\ (e_1, e_2) : \tau_2} \tag{3.10}$$

$$\frac{E \vdash e_1 : \mathcal{L} \qquad E \vdash e_2 : \tau_2 \qquad E \vdash e_3 : \tau_3 \quad \text{if } \tau_2 = \tau_3}{E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau_2} \tag{3.11}$$

FIG. 3.2 – Typage pour la partie fonctionnelle

$$\frac{E \vdash e : \mathcal{L}}{E \vdash \mathbf{mkpar}\ e : \mathcal{G}} \tag{3.12}$$

$$\frac{E \vdash e_1 : \mathcal{G} \qquad E \vdash e_2 : \mathcal{G} \qquad \text{where } \mathbf{Par\_op} = \mathbf{apply} \text{ or } \mathbf{get}}{E \vdash \mathbf{Par\_op}\ e_1\ e_2 : \mathcal{G}} \tag{3.13}$$

$$\frac{E \vdash e_1 : \mathcal{G} \qquad E \vdash e_2 : \mathcal{L} \qquad E \vdash e_3 : \mathcal{G} \qquad E \vdash e_4 : \mathcal{G}}{E \vdash \mathbf{if}\ e_1\ \mathbf{at}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 : \mathcal{G}} \tag{3.14}$$

$$\frac{\forall i (E \vdash e_i : \mathcal{L})}{E \vdash \langle\ e_0\ , \ldots ,\ e_{p-1}\ \rangle : \mathcal{G}} \tag{3.15}$$

FIG. 3.3 – Typage pour les opérateurs parallèles

v ::=	c		(constantes)
	$\mid$	op	(opérateurs)
	$\mid$	$\mathbf{fun}\ x : \mathcal{L}\ \rightarrow\ e$	(valeur fonctionnelle locale)
	$\mid$	$\mathbf{fun}\ x : \mathcal{G}\ \rightarrow\ e$	(valeur fonctionnelle globale)
	$\mid$	(v, v)	(paires de valeurs)
	$\mid$	$\langle\ v\ , \ldots ,\ v\ , \ldots ,\ v\ \rangle$	(vecteur parallèle énuméré)

Les règles d'évaluation classiques et des opérateurs parallèles sont présentées ci-dessous. La primitive de communication étant le **get**.

Dans tous ce qui suit, on note que $e_1[x \leftarrow e_2]$ la substitution des occurrences libres de x dans e_1 par e_2.

Les règles pour les constantes, les opérateurs arithmétiques sont :

$$c \quad \triangleright \quad c \tag{3.16}$$

$$op \quad \triangleright \quad op \tag{3.17}$$

Les deux règles pour les fonctions et les liaisons sont :

$$\mathbf{fun}\ x : \tau\ \rightarrow\ e \quad \triangleright \quad \mathbf{fun}\ x : \tau\ \rightarrow\ e \tag{3.18}$$

$$\frac{e_1 \triangleright v_1 \quad e_2[x \leftarrow v_1] \triangleright v}{\mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 \triangleright v} \tag{3.19}$$

Les cinq règles suivantes concernent les applications :

$$\frac{e_1 \triangleright + \quad e_2 \triangleright (n_1, n_2) \quad n = n_1 + n_2}{(e_1\ e_2) \triangleright n} \tag{3.20}$$

$$\frac{e_1 \rhd \textbf{fix}\ e_2 \rhd (\textbf{fun}\ x : \tau\ \rightarrow\ e_3)\ e_3[x \leftarrow \textbf{fix}(e_2)] \rhd v\ e_2 : \tau}{(e_1\, e_2) \rhd v} \tag{3.21}$$

$$\frac{e_1 \rhd \textbf{fix} \quad e_2 \rhd op}{(e_1\, e_2) \rhd op} \tag{3.22}$$

$$\frac{e_1 \rhd (\textbf{fun}\ x : \tau\ \rightarrow\ e) \quad e_2 \rhd v_2 \quad e[x \leftarrow v_2] \rhd v \quad e_2 : \tau}{(e_1\ e_2) \rhd v} \tag{3.23}$$

$$\frac{e_1 \rhd \langle\ v'_0\ , \ldots,\ v'_{p-1}\ \rangle \quad e_2 \rhd \langle\ v"_0\ , \ldots,\ v"_{p-1}\ \rangle\ avec\ v = v'v"}{\textbf{apply}\, e_1\, e_2 \rhd \langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle} \tag{3.24}$$

Les trois règles suivantes concernent les couples dont (3.26) et (3.27) réalisent la projection :

$$\frac{e_1 \rhd v_1 \quad e_2 \rhd v_2}{(e_1, e_2) \rhd (v_1, v_2)} \tag{3.25}$$

$$\frac{e \rhd (v_1, v_2)}{\textbf{fst}\ e \rhd v_1} \tag{3.26}$$

$$\frac{e \rhd (v_1, v_2)}{\textbf{snd}\ e \rhd v_2} \tag{3.27}$$

Les deux règles suivantes concernent la conditionnelle :

$$\frac{e_1 \rhd \textbf{true} \quad e_2 \rhd v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rhd v} \tag{3.28}$$

$$\frac{e_1 \rhd \textbf{false} \quad e_3 \rhd v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rhd v} \tag{3.29}$$

La règle de l'opérateur de création de vecteurs parallèle est :

$$\frac{e_1 \rhd v \quad \forall i\ (v\ i) \rhd v_i}{\textbf{mkpar}\ \ e_1 \rhd \langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle} \tag{3.30}$$

Les règles de communication sont données par les deux trois suivantes :

$$\frac{e_1 \rhd \langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle \quad e_2 \rhd \langle\ i_0\ , \ldots,\ i_{p-1}\ \rangle}{\textbf{get}\ e_1\ e_2 \rhd \langle v_{i_0 \% p}, \ldots, v_{i_{p-1} \% p} \rangle} \tag{3.31}$$

$$\frac{e_1 \rhd \langle \ldots, \overbrace{\textbf{true}}^{n}, \ldots \rangle \quad e_2 \rhd n \quad e_3 \rhd v_3}{\textbf{if}\, e_1\, \textbf{at}\, e_2\, \textbf{then}\, e_3\, \textbf{else}\, e_4 \rhd v_3} \tag{3.32}$$

$$\frac{e_1 \rhd \langle \ldots, \overbrace{\textbf{false}}^{n}, \ldots \rangle \quad e_2 \rhd n \quad e_4 \rhd v_4}{\textbf{if}\, e_1\, \textbf{at}\, e_2\, \textbf{then}\, e_3\, \textbf{else}\, e_4 \rhd v_4} \tag{3.33}$$

3.2 MSPML avec juxtaposition

Pour évaluer deux programmes parallèles sur la même machine on a ajouter l'opérateur de composition parallèle spatiale appelé la juxtaposition.

3.2.1 Syntaxe et typage

Le noyau que nous considérons ici est présenté dans la figure 3.4 est le même que celui de la figure 3.1, avec l'ajout de la primitive de la composition parallèle **juxta**.

e'	::=	x	(variables)
	|	c	(constantes)
	|	**fun** $x : \tau \ \rightarrow \ e'$	(abstraction)
	|	**let** $x : \tau = e'$ **in** e'	(liaison)
	|	op	(opérateurs)
	|	$(e' \ e')$	(application)
	|	(e', e')	(paires)
	|	**if** e' **then** e' **else** e'	(conditionnelle)
	|	**fst** e'	(premier de la paire)
	|	**snd** e'	(deuxième de la paire)
	|	**mkpar** e'	(vecteur parallèle)
	|	**apply** $e' \ e'$	(application parallèle)
	|	**get** $e' \, e'$	(communication)
	|	**if** e' **at** e' **then** e' **else** e'	(conditionnelle globale)
	|	**juxta** $e' \ e' \ e'$	(composition parallèle

FIG. 3.4 – La syntaxe du noyau de MSPML

L'ajout de la juxtaposition fausse notre sémantique à grand pas d'où la nécessité de la modifier en ajoutant un environnement pour les variables globales dont le mécanisme est décrit ci-dessous dans la section 3.2.2.

Typage

Dans la sémantique à grands pas sans juxtaposition, ici, on fera la distinction entre les expressions locales et expressions globales.

Le typage est le même que celui de la section 3.1.1. La juxtaposition est donnée par la règle suivante :

$$\frac{E \vdash e_1 : \mathcal{L} \qquad E \vdash e_2 : \mathcal{G} \qquad E \vdash e_3 : \mathcal{G}}{E \vdash \textbf{juxta} \ e_1 \ e_2 \ e_3 : \mathcal{G}} \tag{3.34}$$

Proposition 1 (Sûreté du typage) *Soit e une expression MSPML et E un environnement de typage. Si $E \vdash e : \tau$ et $e \rhd v$ alors $E \vdash v : \tau$*

3.2.2 Règles d'évaluation

Les valeurs résultantes des évaluations sont les mêmes que celles définies dans la section 3.1.2

Évaluations locales

L'évaluation $\rhd_{loc}$ des expressions locales est classique (voir la Figure 3.5).

La figure 3.5 contient respectivement les règles d'évaluation des constantes, opérateurs arithmétiques, conditionnelle, projection, paire, opérateur du point fixe et enfin l'addition arithmétique, l'abstraction, la liaison et l'application locale.

Évaluations globales

L'évaluation globale $\rhd_{glo}$ des expressions globales utilise un environnement pour les variables globales : un vecteur parallèle peut être lié à une variable dans un réseau à p processeurs puis cette variable peut être utilisée dans un sous-réseau à p' processeurs avec $p' < p$. Dans ce cas seules p' valeurs sont intéressantes, partant du processeur f dans le réseau à p processeurs renommé 0 dans le sous-réseau.

Ainsi l'évaluation globale dépend du nombre de processeurs dans le sous-réseau courant ainsi que du nom dans le réseau englobant du premier processeur de ce sous-réseau courant.

Les jugements de la sémantique ont la forme $\mathcal{E}_g \vdash_f^{p'} e \rhd_{\text{glob}} v$ qui signifie :

"Dans l'environnement global $\mathcal{E}$, sur le sous-réseau à p' processeurs dont le premier processeur est numéroté f dans le réseau entier, le terme global e s'évalue en v (variable globale)".

L'environnement $\mathcal{E}_{glo}$ ne lie pas simplement une variable et une valeur comme c'est le cas habituellement. Lorsque la valeur liée est un vecteur parallèle, elle l'est avec le numéro dans le réseau entier du premier processeur du sous-réseau courant au moment de mise dans l'environnement. Nous notons un environnement comme une liste d'association et aussi comme une fonction qui si elle est appliquée à une variable retourne la première valeur ou couple entier/valeur associée à cette variable. On a également besoin de remplacer les valeurs *abstractions globales* par un nouveau genre de valeur : les fermetures $[(\textbf{fun}\ x : \tau\ \rightarrow\ e), \mathcal{E}]$, exprimée par la règle suivante :

$$\mathcal{E}_g \vdash_f^{p'} (\textbf{fun}\ x : \mathcal{G}\ \rightarrow\ e) \rhd_{\text{glob}} [(\textbf{fun}\ x : \mathcal{G}\ \rightarrow\ e), \mathcal{E}_g] \tag{3.47}$$

Les deux premières règles indiquent comment retrouver une valeur dans un environnement global. On a deux cas selon que la valeur est un vecteur parallèle (3.48) ou une fermeture globale (3.49).

$$c \quad \triangleright_{loc} \quad c \tag{3.35}$$

$$op \quad \triangleright_{loc} \quad op \tag{3.36}$$

$$\frac{e_1 \triangleright_{loc} \mathbf{true} \quad e_2 \triangleright_{loc} v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \triangleright_{loc} v} \tag{3.37}$$

$$\frac{e_1 \triangleright_{loc} \mathbf{false} \quad e_3 \triangleright_{loc} v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \triangleright_{loc} v} \tag{3.38}$$

$$\frac{e \triangleright_{loc} (v_1, v_2)}{\mathbf{fst}\ e \triangleright_{loc} v_1} \tag{3.39}$$

$$\frac{e \triangleright_{loc} (v_1, v_2)}{\mathbf{snd}\ e \triangleright_{loc} v_2} \tag{3.40}$$

$$\frac{e_1 \triangleright_{loc} v_1 \quad e_2 \triangleright_{loc} v_2}{(e_1, e_2) \triangleright_{loc} (v_1, v_2)} \tag{3.41}$$

$$\frac{e_1 \triangleright_{loc} \mathbf{fix} \quad e_2 \triangleright_{loc} op}{(e_1\ e_2) \triangleright_{loc} op} \tag{3.42}$$

$$\frac{e_1 \triangleright_{loc} + \quad e_2 \triangleright_{loc} (n_1, n_2) \quad n = n_1 + n_2}{(e_1\ e_2) \triangleright_{loc} n} \tag{3.43}$$

$$\mathbf{fun}\ x : \mathcal{L}\ \rightarrow\ e \quad \triangleright_{loc} \quad \mathbf{fun}\ x : \mathcal{L}\ \rightarrow\ e \tag{3.44}$$

$$\frac{e_1 \triangleright_{loc} v_1 \quad e_2[x \leftarrow v_1] \triangleright_{loc} v \qquad e_1 : \mathcal{L}\ e_2 : \mathcal{L}}{\mathbf{let}\ x : \mathcal{L} = e_1\ \mathbf{in}\ e_2 \triangleright_{loc} v} \tag{3.45}$$

$$\frac{e_1 \triangleright_{loc} \mathbf{fix} \quad e_2 \triangleright_{loc} (\mathbf{fun}\ x : \mathcal{L}\ \rightarrow\ e_3) \quad e_3[x \leftarrow \mathbf{fix}(e_2)] \triangleright_{loc} v \quad e_2 : \mathcal{L}}{(e_1\ e_2) \triangleright_{loc} v} \tag{3.46}$$

FIG. 3.5 – règles pour les opérateurs fonctionnelles

La seconde est classique. La première signifie qu'une variable qui a été liée à un vecteur parallèle dans un réseau avec p_1 processeurs et avec comme premier processeur f_1 peut être utilisée dans un réseau avec p_2 processeurs ($p_2 < p_1$) et ayant f_2 comme premier processeur ($f_1 \leq f_2$). Dans ce cas seules p_2 composantes du vecteur sont utilisées partant de la composante numéro $f_2 - f_1$.

$$\frac{\mathcal{E}(x:\mathcal{G}) = (f_1, \langle\ v_0\ , \ldots,\ v_{p_1-1}\ \rangle)}{\mathcal{E}\ \vdash^{p_2}_{f_2}\ x:\mathcal{G}\ \ \rhd_{\text{glob}}\ \ \langle\ v_{f_2-f_1}\ , \ldots,\ v_{f_2-f_1+p_2-1}\ \rangle} \tag{3.48}$$

$$\frac{\mathcal{E}(x:\mathcal{G}) = (f_1, [(\textbf{fun}\ x:\tau\ \rightarrow\ e), \mathcal{E}])}{\mathcal{E}\ \vdash^{p_2}_{f_2}\ x:\mathcal{G}\ \ \rhd_{\text{glob}}\ \ [(\textbf{fun}\ x:\tau\ \rightarrow\ e), \mathcal{E}]} \tag{3.49}$$

La liaison globale est exprimée par les deux règles suivantes :

$$\frac{\begin{array}{l}\mathcal{E}_g\ \vdash^{p'}_f\ e_1\ \ \rhd_{\text{glob}}\ \ v_1\quad e_1:\mathcal{G}\quad e_2:\mathcal{G}\\ (x,(f,v_1))::\mathcal{E}_g\ \vdash^{p'}_f\ e_2\ \ \rhd_{\text{glob}}\ \ v_2\end{array}}{\mathcal{E}_g\ \vdash^{p'}_f\ \textbf{let}\ x:\mathcal{G} = e_1\ \textbf{in}\ e_2\ \ \rhd_{\text{glob}}\ \ v_2} \tag{3.50}$$

$$\frac{\begin{array}{l}\mathcal{E}_g\ \vdash^{p'}_f\ e_1\ \ \rhd_{\text{glob}}\ \ v_1\quad e_1:\mathcal{L}\quad e_2:\mathcal{G}\\ \mathcal{E}_g\ \vdash^{p'}_f\ e_2[x \leftarrow v_1]\ \ \rhd_{\text{glob}}\ \ v_2\end{array}}{\mathcal{E}_g\ \vdash^{p'}_f\ \textbf{let}\ x:\mathcal{L} = e_1\ \textbf{in}\ e_2\ \ \rhd_{\text{glob}}\ \ v_2} \tag{3.51}$$

L'évaluation d'un **mkpar** dépend maintenant du nombre de processeurs dans le sous-réseau :

$$\frac{\forall i \in (0...p'-1),\ e\ i\ \rhd_{\text{loc}}\ v_i}{\mathcal{E}_g\ \vdash^{p'}_f\ \textbf{mkpar}\ e\ \ \rhd_{\text{glob}}\ \ \langle\ v_0\ , \ldots,\ v_{p'-1}\ \rangle} \tag{3.52}$$

Les quatre règles suivantes concernent les applications :

$$\frac{\begin{array}{l}\mathcal{E}_g\ \vdash^{p'}_f\ e_1\ \ \rhd_{\text{glob}}\ \ [(\textbf{fun}\ x:\ \rightarrow\ e), \mathcal{E}]\\ \mathcal{E}_g\ \vdash^{p'}_f\ e_2\ \ \rhd_{\text{glob}}\ \ v_2\quad e_1:\mathcal{G}\quad e_2:\mathcal{G}\\ (x,(f,v_2))::\mathcal{E}\ \vdash^{p'}_f\ e\ \ \rhd_{\text{glob}}\ \ v_1\end{array}}{\mathcal{E}_g\ \vdash^{p'}_f\ (e_1\ e_2)\ \ \rhd_{\text{glob}}\ \ v_1} \tag{3.53}$$

$$\frac{\begin{array}{l}\mathcal{E}_g\ \vdash^{p'}_f\ e_1\ \ \rhd_{\text{glob}}\ \ [(\textbf{fun}\ x:\mathcal{G}\ \rightarrow\ e), \mathcal{E}']\\ e_2 \rhd_{\text{loc}} v_2\ \ \mathcal{E}'\ \vdash^{p'}_f\ e[x \leftarrow v_2]\ \ \rhd_{\text{glob}}\ \ v\\ e_1:\mathcal{G}\quad e_2:\mathcal{L}\end{array}}{\mathcal{E}_g\ \vdash^{p'}_f\ (e_1\ e_2)\ \ \rhd_{\text{glob}}\ \ v} \tag{3.54}$$

$$\frac{e_1 \rhd_{loc} \mathbf{fix} \quad \mathcal{E}' \vdash_f^{p'} e_2 \rhd_{\text{glob}} [(\mathbf{fun}\ x : \mathcal{G} \rightarrow e_3), \mathcal{E}'] \quad \mathcal{E}' \vdash_f^{p'} e_3[x \leftarrow \mathbf{fix}(e_2)] \rhd_{\text{glob}} v \quad e_2 : \mathcal{G}}{\mathcal{E}_g \vdash_f^{p'} (e_1\ e_2) \rhd_{\text{glob}} v} \tag{3.55}$$

$$\frac{\mathcal{E}_g \vdash_f^{p'} e_1 \rhd_{\text{glob}} \langle\ v'_0, \ldots, v'_{p'-1}\ \rangle \quad \mathcal{E}_g \vdash_f^{p'} e_2 \rhd_{\text{glob}} \langle\ v"_0, \ldots, v"_{p'-1}\ \rangle \quad \forall i \in 0...(p'-1),\ v'_i\ v"_i \rhd_{\text{loc}} v_i}{\mathcal{E}_g \vdash_f^{p'} (\mathbf{apply}\, e_1\, e_2) \rhd_{\text{glob}} \langle\ v_0, \ldots, v_{p'-1}\ \rangle} \tag{3.56}$$

Les trois règles suivantes concernent les primitives de communication :

$$\frac{\mathcal{E}_g \vdash_f^{p'} e_1 \rhd_{\text{glob}} \langle\ v_0, \ldots, v_{p'-1}\ \rangle \quad \mathcal{E}_g \vdash_f^{p'} e_2 \rhd_{\text{glob}} \langle\ n_0, \ldots, n_{p'-1}\ \rangle}{\mathcal{E}_g \vdash_f^{p'} (\mathbf{get}\ e_1\ e_2) \rhd_{\text{glob}} \langle\ v_{n0}, \ldots, v_{n(p'-1)}\ \rangle} \tag{3.57}$$

$$\frac{\mathcal{E}_g \vdash_f^{p'} e_1 \rhd_{\text{glob}} \langle\ v_0, \ldots, v_{p'-1}\ \rangle \quad \mathcal{E}_g \vdash_f^{p'} e_2 \rhd_{\text{glob}} v_2 \quad e \rhd_{\text{loc}} n\ et\ 0 \leq n < p'\ et\ v_n \equiv \mathbf{true}}{\mathcal{E}_g \vdash_f^{p'} \mathbf{if}\, e_1\, \mathbf{at}\, e\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \rhd_{\text{glob}} v_2} \tag{3.58}$$

$$\frac{\mathcal{E}_g \vdash_f^{p'} e_1 \rhd_{\text{glob}} \langle\ v_0, \ldots, v_{p'-1}\ \rangle \quad \mathcal{E}_g \vdash_f^{p'} e_2 \rhd_{\text{glob}} v_2 \quad e \rhd_{\text{loc}} n\ et\ 0 \leq n < p'\ et\ v_n \equiv \mathbf{false}}{\mathcal{E}_g \vdash_f^{p'} \mathbf{if}\, e_1\, \mathbf{at}\, e\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \rhd_{\text{glob}} v_2} \tag{3.59}$$

$$\frac{\mathcal{E}_g \vdash_f^{m} e_1 \rhd_{\text{glob}} \langle\ v_0, \ldots, v_{m-1}\ \rangle \quad \mathcal{E}_g \vdash_{f+m}^{p'-m} e_2 \rhd_{\text{glob}} \langle\ v_m, \ldots, v_{p'-1}\ \rangle \quad e \rhd_{\text{loc}} m\ et\ 0 < m < p'}{\mathcal{E}_g \vdash_f^{p'} \mathbf{juxta}\ e_1\ e_2\ e_3 \rhd_{\text{glob}} \langle\ v_0, \ldots, v_{p'-1}\ \rangle} \tag{3.60}$$

3.2.3 Confluence

La confluence de cette sémantique à grand pas avec juxtaposition est vérifiée si, dans le cas où on peut effectuer deux réductions sur un terme, alors il existe un terme qui peut être atteint par une ou plusieurs dérivations à partir des deux termes obtenus.

La confluence de la réduction $\triangleright_{loc}$ locale est indépendante de $\triangleright_{glo}$ la réduction globale, car rappelons que le système de typage utilisé ne permet pas d'avoir un résultat de type supérieur à celui du terme réduit. Dans ce cas on est sûr que la réduction locale de tous terme local ne peut donner qu'un terme local. Par contre la réduction globale peut donner un terme de type inférieur et donc un type local, or une fois la confluence de la réduction locale prouvée, on est sûr que la réduction locale des termes locaux résultants d'une réduction globale est forcément confluente. Ce qui reste à prouver dans ce cas est la confluence de la réduction globale.

La proposition suivante concerne la confluence de la réduction locale :

Proposition 2 *Soit e une expression MSPML close et bien formée. Si* $e \triangleright_{loc} v_1$ *et* $e \triangleright_{loc} v_2$, *alors* $v_1 = v_2$.

La proposition suivante concerne la confluence de la réduction globale :

Proposition 3 *Soit e une expression MSPML close et bien formée. Si* $\mathcal{E} \vdash_f^p e_1 \triangleright_{glob} v_1$ *et* $\mathcal{E} \vdash_f^p e_2 \triangleright_{glob} v_2$, *alors* $v_1 = v_2$.

Les preuves de ces propositions sont données par induction dans l'annexe A.

Chapitre 4

MSPML : Sémantique à petits pas

La sémantique à grand pas de MSPML ne donne pas les étapes de calcul mais juste le résultat. Par conséquent, tous les opérateurs parallèles apparaissent synchrones dans cette sémantique. Pour montrer comment la désynchronisation est traitée dans MSPML, une sémantique distribuée, qui donne les étapes de réduction vers une valeur, est nécessaire. L'évaluation distribuée peut être définie en deux étapes :

1. l'évaluation locale des expressions ;
2. l'évaluation globale des termes distribués.

4.1 MSPML sans juxtaposition

Avant d'introduire la sémantique de MSPML avec juxtaposition, voyons rapidement un aperçu de la sémantique de MSPML sans cette opération de composition parallèle.

4.1.1 Syntaxe et typage

La syntaxe considérée ici est identique à celle présentée dans la figure 4.1. La primitive **juxta** et $\|e_d\|$[1] étant bien évidement exclue.

$\overrightarrow{e_d}$ représente les vecteurs parallèles énumérés,qui sont générés au cours de l'évaluation. **request** est une opération permettant de garantir la désynchronisation, elle est utilisée pour récupérer les données lors d'une communication entre deux processus distincts.

Typage

Dans la syntaxe, on utilise le typage des variables locales et globales. Afin de typer les expressions distribuées le système de type présenté dans les figures 3.2

[1]Cette expression signifie que l'expression e_d est dans une sous-machine

et 3.3 sera repris en lui ajoutant une règle de typage du **request** et une autre pour $\overrightarrow{e_d}$ données respectivement par les règles (4.1) et (4.2).

$$\frac{E \vdash e : \mathcal{L}}{E \vdash \overrightarrow{e} : \mathcal{G}} \tag{4.1}$$

$$\frac{E \vdash e_1 : \mathcal{G} \qquad E \vdash e_2 : \mathcal{G}}{E \vdash \mathbf{request}\, e_1\ e_2 : \mathcal{G}} \tag{4.2}$$

4.1.2 Règles d'évaluation

Les règles de la forme $(e_d, \mathcal{E}_c) \ \rightharpoonup_{\mathrm{i}}\ (e'_d, \mathcal{E}'_c)$ peuvent être lues comme suit :" Au processeur i, l'expression e_d dans l'environnement de communcation $\mathcal{E}_c$ est réduite *localement* à l'expression e'_d dans l'environnement de communication $\mathcal{E}'_c$". La réduction globale $\rightarrow$ est une relation sur les expressions distribuées. Le mécanisme des environnements de communication décrit dans le chapitre 2, peut être formalisé comme suit. La sémantique manipule des termes qui sont des paires composées d'une expression MSPML et d'un environnement de communication. L'environnement de communication lui même est une liste composée d'un numéro de m-étape et d'une valeur.

- lors de l'évaluation d'un **get**, la valeur est mise dans l'environnement de communication et le **get** devient un **request**, si la communication n'est pas à destination du processus qui évalue :

$$(\mathbf{get}\, \overrightarrow{v_d}\ \overrightarrow{j}, \mathcal{E}_c,) \ \rightharpoonup_{\mathrm{i}}\ ((\overrightarrow{\mathbf{request}\, (\mathbf{mstep}\ (\mathcal{E}_\mathbf{c}) + \mathbf{1})\mathbf{j}}), ((\mathbf{mstep}(\mathcal{E}_c) + 1), v_d) :: \mathcal{E}_c) \quad si\ j \neq i \tag{4.3}$$

  Sinon il y a simplement mise de la valeur dans l'environnement de communication :

$$(\mathbf{get}\, \overrightarrow{v_d}\ \overrightarrow{i}, \mathcal{E}_c,) \ \rightharpoonup_{\mathrm{i}}\ (\overrightarrow{v_d}, ((\mathbf{mstep}(\mathcal{E}_c) + 1), v_d) :: \mathcal{E}_c) \tag{4.4}$$

- pour que les échanges se fassent, il faut considéré l'évaluation globalement et non plus localement. On manipule alors des vecteurs parallèles de paires (expression MSPML, environnement de communication). L'échange de message entre deux processeurs est modélisé par la règle suivante :

$$\frac{(e_{d_i} = \Gamma[\mathbf{request}\, n\ \ j]) \text{ et } ((n, v_d) \in \mathcal{E}_{c_j}}{\begin{array}{c} \ll\ (e_{d_0}, \mathcal{E}_{c_0})\ , \ldots ,\ (e_{d_i}, \mathcal{E}_{c_i})\ , \ldots ,\ (e_{d_{p-1}}, \mathcal{E}_{c_{p-1}})\ \gg \\ \rightarrow \\ \ll\ (e_{d_0}, \mathcal{E}_{c_0})\ , \ldots ,\ (\Gamma[v_{d_i}], \mathcal{E}_{c_i})\ , \ldots ,\ (e_{d_{p-1}}, \mathcal{E}_{c_{p-1}})\ \gg \end{array}} \tag{4.5}$$

  où Γ est un contexte, semblable aux contextes définis à la figure 4.2.2 (expressions **juxta** et $\|e_d\|$ sont exclus)

Les valeurs obtenues à l'évaluation des termes distribués sont ceux donnés dans la figure 4.2 sans $\|v_d\|$.

4.2 MSPML avec juxtaposition

Encore une fois, l'ajout de la composition parallèle nous a amené à modifier la sémantique à petit pas afin de lui rendre sa confluence. Plusieurs raisons en sont la cause. En particulier, le nombre de processeurs, qui est aussi la taille des vecteurs, n'est plus constant et dépend du contexte d'évaluation du terme.

Les règles de la forme $(e_d, \mathcal{E}_c) \rightharpoonup_{\mathrm{i}} (e'_d, \mathcal{E}'_c)$ peuvent être lues comme suit :“ Au processeur i, l'expression e_d dans l'environnement de communcation $\mathcal{E}_c$ est réduite *localement* à l'expression e'_d dans l'environnement de communication $\mathcal{E}'_c$”.

4.2.1 Syntaxe et typage

La syntaxe est celle présentée dans la figure 4.1. $\|e_d\|$ représente l'expression e_d dans une sous machine.

$$
\begin{array}{rcll}
e_d & ::= & x & \mid\ c \\
 & \mid & \textbf{fun}\ x{:}\mathcal{L} \rightarrow e_d & \mid\ \textbf{fun}\ x{:}\mathcal{G} \rightarrow e_d \\
 & \mid & \textbf{let}\ x{:}\mathcal{L} = e_d\ \textbf{in}\ e_d & \mid\ \textbf{let}\ x{:}\mathcal{G} = e_d\ \textbf{in}\ e_d \\
 & \mid & op & \mid\ (e_d\ e_d) \\
 & \mid & (e_d, e_d) & \mid\ \textbf{if}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\
 & \mid & \textbf{fst}\ e_d & \mid\ \textbf{snd}\ e_d \\
 & \mid & \textbf{mkpar}\ e_d & \mid\ \textbf{apply}\ e_d\ e_d \\
 & \mid & \textbf{get}\ e_d\ e_d & \mid\ \textbf{if}\ e_d\ \textbf{at}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\
 & \mid & \textbf{request}\ e_d\ e_d & \mid\ \overrightarrow{e_d} \\
 & \mid & \|e_d\| & \mid\ \textbf{juxta}\ e_d\ e_d\ e_d
\end{array}
$$

FIG. 4.1 – La syntaxe du noyau du langage

Typage : Le même typage que celui de la section 4.1.1 est utilisé, mais en ajoutant la règle de typage pour $\|e_d\|$ donné par la règle (4.6) :

$$\frac{E \vdash e : \tau}{E \vdash \|e\| : \tau} \tag{4.6}$$

Proposition 4 (Sûreté du typage) *Soit e_d une expression MSPML close et bien formée. Si $E \vdash e : \tau$ et*

$$(e_d, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (v_d, step', abs\_pid', \mathcal{E}'_c, \mathcal{E}'_p, \mathcal{E}'_{pid}, \mathcal{E}'_{step})$$

alors $E \vdash v_d : \tau$

$$v_d \quad ::= \quad \textbf{fun}\ x\ \rightarrow\ e_d \quad | \quad c \quad | \quad op \quad | \quad (v_d,\ v_d) \quad | \quad \overrightarrow{v_d} \quad | \quad \|v_d\|$$

FIG. 4.2 – Les valeurs distribuées

4.2.2 Règles d'évaluation

Les valeurs de la réduction locale sont données dans la figure 4.2. $\|v_d\|$ représente la valeur dans une sous-machine.

Les expressions distribuées sont notées :

$$\ll (e_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}), \ldots, (e_{d_{p-1}}, step_{p-1}, abs\_pid_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \gg$$

et les valeurs associées à ces expressions sont :

$$\ll (v_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}), \ldots, (v_{d_{p-1}}, step_{p-1}, abs\_pid_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \gg$$

Numérotation des m-étapes

Le numéro de m-étape sert à distinguer les messages et à repérer les valeurs stockées à des m-étapes données. Ainsi, les numéros de m-étape doivent être unique.

structure du numéro de m-étape est un couple (num, jux) :

– **num** est de type num_mstep où :

num_mstep = { mstep_juxta :prefix, int_mstep :entier}
prefix = Vide | L **of** prefix | R **of** prefix

– **jux** est un entier indiquant le numéro de juxtaposition.

À chaque communication (appel d'un (**get**) ou d'un (**ifat**) la partie entière du numéro de m-étape est incrémenté de un par la fonction **inc_mstep**. À chaque juxtaposition la fonction **new** remet à zéro la partie entière de la variable step et ajoute à gauche L si $hd(\mathcal{E}_{pid}) < m$ (m est le premier argument de la juxtaposition) ou R sinon. Le numéro de juxtaposition jux est incrémenté de un à chaque juxtaposition, c'est la partie de la variable step qui nous garantie l'unicité des numéros des m-étapes.

Il est possible d'avoir une imbrication de juxtaposition alors dans ce cas le mécanisme est le même. Pour donner une meilleur idée du mécanisme de numérotation on a joint la Figure 4.3.

Mécanisme de communication

À chaque fois une expression **get** $\overrightarrow{v_d}$ $\overrightarrow{j}$, est évaluée à un processus i donné :

1. La partie entière de $step_i$ est incrémentée par un.
2. La valeur v_d contenue dans le vecteur parallèle énuméré $\overrightarrow{v_d}$ est sauvegardée avec la valeur de $step_i$ dans l'environnement de communication. Un environnement

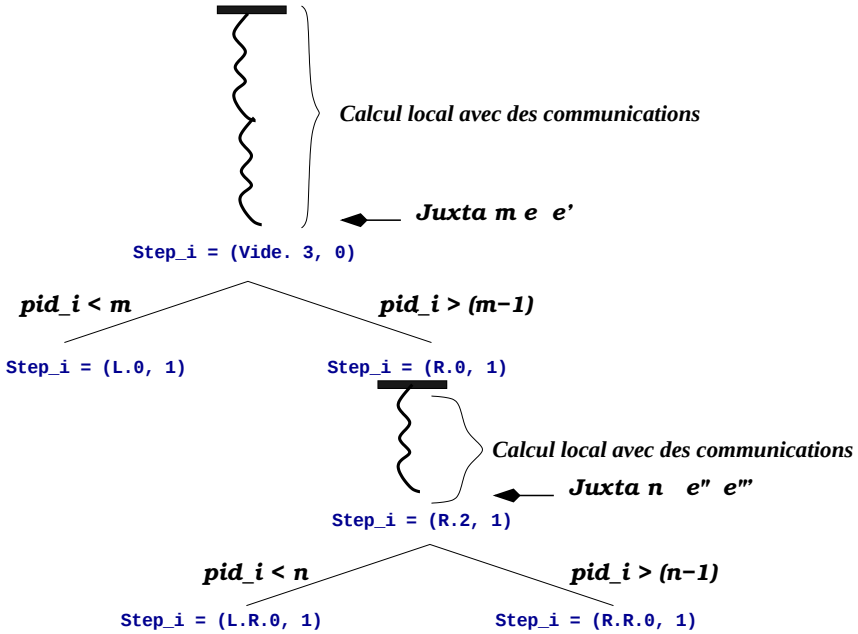

FIG. 4.3 – Numérotation des étapes de communication **mstep** dans une Juxtaposition

de communication peut être vu comme une liste d'association qui relie les nombres de m-étapes avec les valeurs.

3. La valeur j que tient ce processus dans le vecteur parallèle énuméré $\overrightarrow{j}$ est le numéro de processus duquel le processus i veut recevoir une valeur. Ainsi, le processus i envoie une requête au processus abs_pid_j[2] : il demande la valeur à la m-étape $step_i$. Quand le processus j reçoit la requête (des *threads* sont dédiés au traitement de telles requêtes, donc, le travail du processus j n'est pas interrompu), deux cas se présentent :
   - $step_j \geq step_i$: cela veut dire que le processus j a déjà atteint la même m-étape que le processus i. Ainsi, le processus j accède dans son environnement de communication à la valeur associée à la m-étape $step_i$ et l'envoie au processus i.
   - $step_j < step_i$: rien ne peut se faire jusqu'à ce que le processus j atteigne la même m-étape que le processus i.

Si $i = j$, l'étape 3 n'est pas exécutée.

L'ordre sur le numéro de m-étape sur les deux composantes entières $int_m step$ de num et jux est lexicographique inverse. N.B. La relation d'ordre sur les numéros de

[2] abs_pid_j représente l'identifiant du processeur dans la machine absolue. Il est calculée comme suit :$abs\_pid_j = (hd(\mathcal{E}_{pid}) - abs\_pid_i) + (j \% hd(\mathcal{E}_p))$.% représente le modulo utilisé pour la numérotation circulaire.

$$
\begin{array}{rcll}
\Gamma & ::= & [] & | \quad \Gamma\ e_d \\
& | & v_d\ \Gamma & | \quad \textbf{snd}\ \Gamma \\
& | & \textbf{snd}\ \Gamma & | \quad \textbf{let}\ x : \tau = \Gamma\ \textbf{in}\ e_d \\
& | & \overrightarrow{\Gamma} & | \quad \textbf{if}\ \Gamma\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\
& | & \textbf{mkpar}\,\Gamma & | \quad \textbf{apply}\,\Gamma\,e_d \\
& | & \textbf{apply}\,v_d\,\Gamma & | \quad \textbf{get}\,\Gamma\,e_d \\
& | & \textbf{get}\,v_d\,\Gamma & | \quad \textbf{if}\,\Gamma\,\textbf{at}\,e_d\,\textbf{then}\,e_d\,\textbf{else}\,e_d \\
& | & \textbf{if}\,v_d\,\textbf{at}\,\Gamma\,\textbf{then}\,e_d\,\textbf{else}\,e_d & | \quad \|\Gamma\| \\
& | & \textbf{juxta}\ \Gamma\ e_d\ e_d &
\end{array}
$$

FIG. 4.4 – Les contextes d'évaluation de la sémantique distribuée

m-étape est donnée par rapport à la partie entière de la première composante de la variable *step*.

Réduction locale

La réduction locale est peut être subdivisée en trois groupes de règles : les contextes et la règle de contexte, la réduction fonctionnelle qui correspond à une sémantique à petits pas classique et enfin, la réduction pour les opérations parallèles spécifiques à MSPML.

La réduction de tête ne peut être appliquée dans n'importe quel contexte. On rappel que la stratégie choisie est la stratégie faible d'appel par valeur.

Donc, les contextes forcent l'évaluation des arguments avant de permettre la réduction.

Les **contextes** présentés dans la figure 4.2.2 sont nécessaires. Ces contextes sont utilisés avec la règle (4.7).

La définition de contextes permet d'avoir du déterminisme i.e. il n'existe qu'un chemin de réduction possible.

Les règles de la réduction fonctionnelle, ne changent que le premier composant du tuple.

L'environnement $\mathcal{E}_p$ (resp.$\mathcal{E}_{pid}$) est initialisé au début par le nombre de processeurs (resp. l'identifiant absolu *abs_pid*).

La règle (4.17) permet la création de vecteurs parallèles énumérés. $(\overrightarrow{v_d\ i})$ est un vecteur parallèle énuméré qui contient v_d dans le processeur i.

La règle (4.18) est une règle parallèle classique.

Lors de l'évaluation d'un **get**, la valeur est mise dans l'environnement de communication et le **get** devient un **request**, si la communication n'est pas à destination du processus qui évalue alors la règle (4.19) est appliquée. Sinon il y a simplement mise de la valeur dans l'environnement de communication(Règle 4.20). Pour l'autre règle de communication **ifat** le même principe est présenté respectivement dans les règles (4.21) et (4.22).

À l'évaluation d'une **juxta**, la machine absolue est subdivisée en deux sous-machine comme suit :

1. pour l'ensemble des processus de la machine le contenu de la variable *step* est empilé dans l'environnement $\mathcal{E}_{step}$, puis la partie entière du numéro de m-étape est remis à zéro et son numéro de juxtaposition est incrémenté de un.
2. pour les processus dont le $pid < m$ (resp. $pid > m-1$)[3] L (resp. R)est ajouté à gauche du numéro de m-étape. On empile $hd(\mathcal{E}_{pid})$ (resp. $hd(\mathcal{E}_{pid}) - m$) dans l'environnement $\mathcal{E}_{pid}$, et aussi le nouveau numéro de processeurs m (resp. $p-m$) est empilé dans l'environnement $\mathcal{E}_p$.Cela est réalisé par la règle (4.23) (resp. (4.24)).
3. une fois le calcul de $\|e_d\|$ (on obtient donc, $\|v_d\|$) est fini la règle (4.25) est appliquée afin de restaurer l'ancien état de la machine en dépilant les environnements $\mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}$. Le *num* du numéro de la m-étape *step* est remplacé par le contenu du sommet de l'environnement $\mathcal{E}_{step}$ et cela avant de la dépilé.

N.B. Afin de simplifier l'écriture l'incrémentation de la partie entière de *step* n'est pas détaillée. C'est-à-dire l'écriture $(num+1)$ sous entend que l'incrémentation de $step = (num, jux)$ est $step' = (lst.inc\_mstep(n), jux)$ avec $num = lst.(n)$ tel que lst est un prefix et n un entier.

Réduction globale

Le passage au contexte globale se fait par application de la réduction locale réalisé par la règle (4.26).

La seconde règle signifie que si un processeur i demande la valeur détenue par un processeur j à la m-étape n (**request** et que l'environnement de communication $\mathcal{E}_{c_j}$ du processeur j contient la valeur v_d à la m-étape n alors la valeur v_d est envoyé au processeur i. Sinon la règle ne peut pas être appliquée : si le processeur j n'a pas atteint la m-étape numéro n, alors le processeur i doit attendre.

[3] m est le premier argument de la juxtaposition

$$\frac{(e_{d_i}, step_i, abs\_pid_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i}) \rightharpoonup (e'_{d_i}, step'_i, abs\_pid'_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i})}{(\Gamma(e_{d_i}), step_i, abs\_pid_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i}) \rightharpoonup (\Gamma(e'_{d_i}), step'_i, abs\_pid'_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i})} \text{ (La règle de contexte)} \quad (4.7)$$

$$((\textbf{fun } x \ \rightarrow \ e_d)v_d, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (e_d[x \leftarrow v_d], step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p\mathcal{E}_{pid}, \mathcal{E}_{step}) \text{ avec } v_d : \tau \quad (4.8)$$

$$((\textbf{let } x : \tau = v_d \textbf{ in } e_d), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (e_d[x \leftarrow v_d], step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.9)$$

$$(+(n_1, n_2), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (n, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \text{ avec } n = n_1 + n_2 \quad (4.10)$$

$$(\textbf{fst } (v_{d_1}, v_{d_2}), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (v_{d_1}, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.11)$$

$$(\textbf{snd } (v_{d_1}, v_{d_2}), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (v_{d_2}, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.12)$$

$$(\textbf{fix}(\textbf{fun } x : \tau \ \rightarrow \ e_d), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (e_d[x \leftarrow \textbf{fix}(\textbf{fun } x \rightarrow e_d)], step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p\mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.13)$$

$$(\textbf{fix}(\textbf{op}), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (\textbf{op}, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p\mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.14)$$

$$((\textbf{if true then } e_1 \textbf{ else } e_2), step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (e_1, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p\mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.15)$$

$$((\textbf{if False then } \mathbf{e_1} \textbf{ else } \mathbf{e_2}), \mathbf{step}, \mathbf{abs\_pid}, \mathcal{E}_\mathbf{c}, \mathcal{E}_\mathbf{p}, \mathcal{E}_\mathbf{pid}, \mathcal{E}_\mathbf{step}) \rightharpoonup (e_2, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p\mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.16)$$

FIG. 4.5 – Réductions fonctionnelles locales

$$(\mathbf{mkpar}\, v_d, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (\overrightarrow{(v_d\ i)}, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \quad avec\ i = hd(\mathcal{E}_{pid}) \tag{4.17}$$

$$(\mathbf{apply}\, \overrightarrow{v_{d_1}} \overrightarrow{v_{d_2}}, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (\overrightarrow{v_{d_1}\ v_{d_2}}, step, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \tag{4.18}$$

$$\begin{array}{l}(\mathbf{get}\, \overrightarrow{v_d}\ \overrightarrow{j}, (num, jux), abs\_pid, \mathcal{E}_c,\\ \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})\end{array} \rightharpoonup \begin{array}{l}(\overrightarrow{\mathbf{request}\, (num+1, jux)\ \ abs\_pid_j},\\ (num+1, jux), abs\_pid, (((num+1), jux), v_d) :: \mathcal{E}_c,\\ \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})[\mathbf{si}\ hd(\mathcal{E}_{pid}) \neq j]\\ [\mathbf{avec}\ abs\_pid_j = (hd(\mathcal{E}_{pid}) - abs\_pid) + (j \% hd(\mathcal{E}_p))]\end{array} \tag{4.19}$$

$$\begin{array}{l}(\mathbf{get}\, \overrightarrow{v_d}\ \overrightarrow{i}, (num, jux), abs\_pid,\\ \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})\end{array} \rightharpoonup \begin{array}{l}(\overrightarrow{v_d}, (num+1, jux), abs\_pid,\\ ((num+1, jux), v_d) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid},\ \mathcal{E}_{step})\end{array} \tag{4.20}$$

$$\begin{array}{l}(\mathbf{if}\ \overrightarrow{b}\ \mathbf{at}\, n\, \mathbf{then}\, v_1\, \mathbf{else}\, v_2,\\ (num, jux), abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})\end{array} \rightharpoonup \begin{array}{l}(\mathbf{if}\ (\mathbf{request}\, ((num+1), jux)\ \ abs\_pid_n)\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2, step,\\ abs\_pid, ((num+1, jux), b) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})[\mathbf{si}\quad n \neq hd(\mathcal{E}_{pid})]\\ [\mathbf{avec}\ abs\_pid_n = (hd(\mathcal{E}_{pid}) - abs\_pid) + (n \% hd(\mathcal{E}_p)]\end{array} \tag{4.21}$$

$$\begin{array}{l}(\mathbf{if}\ \overrightarrow{b}\ \mathbf{at}\, i\, \mathbf{then}\, v_1\, \mathbf{else}\, v_2, (num, jux), abs\_pid,\\ \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})\end{array} \rightharpoonup \begin{array}{l}(\mathbf{if}\ b\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2, (num, jux), abs\_pid,\\ (((num+1), jux), b) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})\end{array} \tag{4.22}$$

FIG. 4.6 – Les règles de réductions locales parallèles

$$(\textbf{juxta}\ m\ e_{d_1}\ e_{d_2}, (num, jux), abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (\|e_{d_1}\|, (l.new(n), jux+1), abs\_pid, \mathcal{E}_c, m :: \mathcal{E}_p, hd(\mathcal{E}_{pid}) :: \mathcal{E}_{pid}, step :: \mathcal{E}_{step})[\text{si } pid < m \text{ avec } pid = hd(\mathcal{E}_{pid})] \quad (4.23)$$

$$(\textbf{juxta}\ m\ e_{d_1}\ e_{d_2}, (num, jux), abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (\|e_{d_2}\|, (r.new(n), jux+1), abs\_pid, \mathcal{E}_c, p-m :: \mathcal{E}_p, hd(\mathcal{E}_{pid}) - m :: \mathcal{E}_{pid}, step :: \mathcal{E}_{step}) [\text{si } pid > p-m \text{ avec } pid = hd(\mathcal{E}_{pid})] \quad (4.24)$$

$$(\|v_d\|, step, abs\_pid, \mathcal{E}_c, p' :: \mathcal{E}_p, pid :: \mathcal{E}_{pid}, step' :: \mathcal{E}_{step}) \rightharpoonup (v_d, step', abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \quad (4.25)$$

FIG. 4.7 – la règle de réduction parallèle de la juxtaposition

$$\frac{(e_{d_i}, step_i, abs\_pid_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i}) \ \rightharpoonup \ (e'_{d_i}, step'_i, abs\_pid'_i, \mathcal{E'}_{c_i}, \mathcal{E'}_{p_i}, \mathcal{E'}_{pid_i}, \mathcal{E'}_{step_i})}{\begin{array}{c} \ll \ (e_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}) \ , \ldots , \ (e_{d_i}, step_i, abs\_pid_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i}) \ , \ldots , \\ (e_{d_{p-1}}, step_{p-1}, abs\_pid_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \ \gg \\ \rightarrow \\ \ll \ (e_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}) \ , \ldots , \ (e'_{d_i}, step'_i, abs\_pid'_i, \mathcal{E'}_{c_i}, \mathcal{E'}_{p_i}, \mathcal{E'}_{pid_i}, \mathcal{E'}_{step_i}) \ , \ldots , \\ (e_{d_{p-1}}, step_{p-1}, abs\_pid_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \ \gg \end{array}} \tag{4.26}$$

$$\frac{(e_{d_i} = \Gamma[\textbf{request}\, n \ \ j]) \text{ et } ((n, v_d) \in \mathcal{E}_{c_j}}{\begin{array}{c} \ll \ (e_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}) \ , \ldots , \ (e_{d_i}, step_i, abs\_pid_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i}) \ , \ldots , \\ (e_{d_{p-1}}, step_{p-1}, abs\_pid_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \ \gg \\ \rightarrow \\ \ll \ (e_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}) \ , \ldots , \ (\Gamma[v_{d_i}], step_i, abs\_pid_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{step_i}) \ , \ldots , \\ (e_{d_{p-1}}, step_{p-1}, abs\_pid_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \ \gg \end{array}} \tag{4.27}$$

FIG. 4.8 – Les règles de réductions globales

4.2.3 Exemple

Soit l'expression $E \equiv \textbf{juxta}\ 2\ E_1\ E_2$ tel que :

$E_1 \equiv \textbf{get}\ this\ this$
$E_2 \equiv \textbf{get}(\textbf{mkpar}\,(\textbf{fun}\ i\ \rightarrow\ i+1))(\textbf{mkpar}\,(\textbf{fun}\ i\ \rightarrow\ i-1))$
$this \equiv \textbf{mkpar}\,(\textbf{fun}\ pid\ \rightarrow\ pid)$
Pour cet exemple, on prend le nombre de processeurs $p = 4$.

Pour $Abs_i \in [0,1]$,
$(\textbf{juxta}\ 2\ E_1\ E_2, (0,0), Abs_i, [], [4], [Abs_i], []) \xrightarrow{(4.23)}$

$(\|\textbf{get}\ this\ this\|, (L.0,1), Abs_i, [], [2;4], [Abs_i; Abs_i], [0]) \xrightarrow{(4.17)}$

$(\|\textbf{get}(\overrightarrow{(\textbf{fun}\ pid\ \rightarrow\ pid)\ Abs_i})\ this\|, (L.0,1), Abs_i, [], [2;4], [Abs_i; Abs_i], [0]) \xrightarrow{(4.8)}$

$(\|\textbf{get}(\overrightarrow{Abs_i})\ this\|, (L.0,1), Abs_i, [], [2;4], [Abs_i; Abs_i], [0]) \xrightarrow{(4.17)(4.8)}$

$(\|\textbf{get}\overrightarrow{Abs_i}\ \overrightarrow{Abs_i}\|, (L.0,1), Abs_i, [], [2;4], [Abs_i; Abs_i], [0]) \xrightarrow{(4.20)}$

$(\|\overrightarrow{Abs_i}\|, (L.1,1), Abs_i, [((L.1,1), pid)], [2;4], [Abs_i; Abs_i], [0]) \xrightarrow{(4.25)}$

$(\overrightarrow{Abs_i}, (0,1), Abs_i, [((L.1,1), pid)], [4], [Abs_i], [])$

L'application de la règle (4.26) au niveau du processeur 0 on obtient :

$$(\overrightarrow{0}, (0,1), 0, [((L.1,1),0)], [4], [0], [])\ \ldots\ \text{(I)}$$

L'application de la règle (4.26) au niveau du processeur 1 on obtient :

$$(\overrightarrow{1}, (0,1), 1, [((L.1,1),1)], [4], [1], [])\ \ldots\ \text{(II)}$$

Pour $Abs_i = 2$,
$(\textbf{juxta}\ 2\ E_1\ E_2, (0,0), Abs_i, [], [4], [Abs_i], []) \xrightarrow{(4.24)}$

$(\|\textbf{get}(\textbf{mkpar}\,(\textbf{fun}\ pid\ \rightarrow\ pid+1))(\textbf{mkpar}\,(\textbf{fun}\ pid\ \rightarrow\ pid-1))\|, (R.0,1), Abs_i, [], [2;4], [(Abs_i-2); Abs_i], [0]) \xrightarrow{(4.17)}$

$(\|\textbf{get}(\textbf{fun}\ pid\ \rightarrow\ pid+1 Abs_i)\textbf{mkpar}\,(\textbf{fun}\ pid\ \rightarrow\ pid-1)\|, (R.0,1), Abs_i, [], [2;4], [(Abs_i-2); Abs_i], [0]) \xrightarrow{(4.8)}$

$(\|\textbf{get}(\overrightarrow{Abs_i+1})\textbf{mkpar}\,(\textbf{fun}\ pid\ \rightarrow\ pid-1)\|, (R.0,1), Abs_i, [], [2;4], [(Abs_i-2); Abs_i], [0]) \xrightarrow{(4.17)(4.8)}$

$(\|\textbf{get}(\overrightarrow{Abs_i+1})(\overrightarrow{Abs_i-1})\|, (R.0,1), Abs_i, [], [2;4], [(Abs_i-2); Abs_i], [0]) \xrightarrow{(4.19)}$

$(\|\overrightarrow{\textbf{request}\,(R.1,1)\ \ (abs\_pid(Abs_i-3))}\|, (R.1,1), Abs_i, [((R.1,1), Abs_i-1)], [2;4], [(Abs_i-2); Abs_i], [0]) \overset{(4.25)}{\longrightarrow}$

$(\overrightarrow{\textbf{request}\,(R.1,1)\ \ (abs\_pid(Abs_i-3))}, (0,1), Abs_i, [((R.1,1), Abs_i-1)], [4], [Abs_i], [])$

la valeur $abs\_pid(Abs_i-3)$ est calculée comme suit :$2+(Abs_i-3)\%2$ L'application de la règle (4.27) au niveau du processeur 2 on obtient :

$$(\overrightarrow{2}, (0,1), 0, [((R.1,1),1)], [4], [2], []) \text{ ... (III)}$$

L'application de la règle (4.27) au niveau du processeur 3 on obtient :

$$(\overrightarrow{1}, (0,1), 0, [((R.1,1),2)], [4], [3], []) \text{ ... (IV)}$$

L'état de la machine est comme suit :

$$\ll \text{(I), (II), (III), (IV)} \gg$$

4.2.4 Confluence

La réduction locale de la sémantique à petit pas avec juxtaposition nous conduit soit à une valeur 4.2 soit à une expression avec un **request**. Sa confluence est exprimée par le lemme 1

Lemme 1 (Confluence de la réduction locale) *Soit e_d une expression MSPML close et bien formée. Si*

$$(e_d, mstep, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (e'_d, mstep, abs\_pid', \mathcal{E}'_c, \mathcal{E}'_p, \mathcal{E}'_{pid}, \mathcal{E}'_{step})$$

et

$$(e_d, mstep, abs\_pid, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (e''_d, mstep, abs\_pid'', \mathcal{E}''_c, \mathcal{E}''_p, \mathcal{E}''_{pid}, \mathcal{E}''_{step})$$

Alors, il existe $(e'''_d, mstep, abs\_pid''', \mathcal{E}'''_c, \mathcal{E}'''_p, \mathcal{E}'''_{pid}, \mathcal{E}'''_{step})$ *tel que :*

$$(e'_d, mstep, abs\_pid', \mathcal{E}'_c, \mathcal{E}'_p, \mathcal{E}'_{pid}, \mathcal{E}'_{step}) \rightharpoonup (e'''_d, mstep, abs\_pid''', \mathcal{E}'''_c, \mathcal{E}'''_p, \mathcal{E}'''_{pid}, \mathcal{E}'''_{step})$$

et

$$(e''_d, mstep, abs\_pid'', \mathcal{E}''_c, \mathcal{E}''_p, \mathcal{E}''_{pid}, \mathcal{E}''_{step}) \rightharpoonup (e'''_d, mstep, abs\_pid''', \mathcal{E}'''_c, \mathcal{E}'''_p, \mathcal{E}'''_{pid}, \mathcal{E}'''_{step})$$

La réduction globale est indépendante de la réduction locale d'où la nécessité de prouver sa confluence. La confluence de la réduction globale est exprimée par la proposition 5

Proposition 5 (Confluence de la réduction globale) *Soit E une expression MSPML distribuée est un tuple de longueur p.* $E = \ll (e_{d_0}, step_0, abs\_pid_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{step_0}), \ldots, (e_{d_{p-1}}, step_{p-}$ $\mathcal{E}_{pid_{p-1}}, \mathcal{E}_{step_{p-1}}) \gg$.

Si $E \rightarrow E'$ *et* $E \rightarrow E"$, *alors il existe* $E"'$ *tel que :*
$E' \rightarrow E"'$ *et* $E" \rightarrow E"'$

Les preuves correspondantes respectivement au lemme et à la proposition sont données dans l'annexe A.

Chapitre 5

MSPML : Implantation

MSPML est implanté en tant qu'une bibliothèque d'Objective Caml. Il est divisé en deux parties : le noyau de la bibliothèque qui ne contient que les primitives, et la bibliothèque standard implantée en utilisant les primitives du noyau. Le noyau n'utilise que les bibliothèques d'Objective Caml. Les communications sont faites en utilisant le protocole TCP/IP disponible dans le module Unix d'Objective Caml et les valeurs sont linéarisées à l'aide du module Marshal.

Les primitives décrites dans la figure 2.2 sont contenues dans le *module Mspml*. Bien évidement, ils ne sont pas directement implantés en utilisant les modules Unix et Marshal. Le module Mspml utilise notre *module Tcpip* qui offre un petit ensemble de fonctions similaires à celles de MPI (Voir figure 5.1) implantées en utilisant le module Unix d'Objective Caml. Le module *Mspml* est écrit dans le style SPMD. *Le module Tcpip* n'est pas disponible pour l'application du programmeur qui peut juste utiliser *le module Mspml* qui offre les fonctions présentées dans la section 2.2. En utilisant *le module Mspml*, le programmeur ne peut pas écrire des programmes dans le style SPMD et par conséquent il évite les problèmes qui interviennent avec ce paradigme de programmation.

5.1 Le module Tcpip

Les types : **mstep**, **num** et **prefix** sont définis afin de pouvoir implanter la numérotation des m-étapes.

Pour **mstep** et **num** on a utilisé les enregistrements à champs mutables afin de pouvoir manipuler séparément les différentes parties du numéro de la m-étape *step*.

Les environnements

– Les environnements **env_p**, **env_pid** et **env_mstep** sont implantés avec la structure de données pile d'Objective Caml (Stack), permettant de stocker respectivement la valeur p qui représente le nombre de processeurs du réseau, *pid*, L'identifiant du processeur et *step.num_mstep.mstep* le premier champ

type store_mode = Get | Mget
type prefix = Vide | L **of** prefix | R **of** prefix
type num = { **mutable** int_mstep : int; **mutable** juxta_mstep : prefix; }
type mstep = { **mutable** num_mstep : num; **mutable** num_juxta : int; }
val pid : unit → int
val p : unit → int
val g : unit → float
val l : unit → float
val finalize : unit → unit
val initialize : unit → unit
val store : α → **store_mode** → unit
val request : mstep → α
val reset_mstep : unit → unit
val step: mstep
val inc_juxta : mstep → unit
val new_mstep : mstep → int → unit
val env_pid : int Stack.t
val env_p : int Stack.t
val env_mstep:mstep Stack.t
val com_env : num_mstep Hach.t

FIG. 5.1 – Le module Tcpip.

de l'enregistrement mstep. La fonction *push* et *pop* sont utilisés pour empiler et dépiler les piles.
- l'environnement de communication **com_env** est une table de hachage dont la clé de ses éléments est le numéro de la m-étape *step* chaque cellule contient la valeur correspondante à la m-étape *step* désignée par sa clé.

Les fonctions Les fonctions *du module Tcpip* dont les signatures sont données dans la figure 5.1 sont décrites comme suit :
- **p()** et **pid()** donnent respectivement le nombre total de processeurs et d'identifiants de processeurs. Dans cette nouvelle version les fonctions récupèrent leurs valeurs à partir du sommets des environnments $\mathcal{E}_p$ et $\mathcal{E}_{pid}$ décrit ci-dessus.
- **l()** et **g()** donnent respectivement la latence du réseau et le temps nécessaire pour échanger des mots de données entre deux processeurs.
- *initialize* est similaire à MPI_Init. Elle permet de lire les paramètres p, l et g à partir du fichier (HOME)˜ .mspmlrc.
- *finalize* est similaire à MPI_Finalize. Elle permet de nettoyer tous les états de l'exécution.
  Au début de l'exécution de MSPML, il y a deux processus légers ou "threads " par processeurs : un correspond à la réduction locale de la sémantique distribuée (chapitre 4), utilisé pour le programme principal, et l'autre créer par **initialize** et qui s'occupe des communications. Le second répond aux requêtes émises par les autres processeurs. Le second répond aux requêtes émises par les autres processeurs.
- La fonction *new_mstep* permet de mettre à zéro le champ int_mstep de la variable step et d'ajouter le L si le pid du processeur à la juxtaposition est inférieur à m (le nombre de processeurs de la sous machine gauche), R sinon.
- La fonction *inc_juxta* permet d'incrémenter le champ **num_juxta** à chaque opération de juxtaposition.
- La fonction *reset_mstep* Permet de vérifier à l'incrémentation de la variable step si le maximum de m-étapes est atteint. Elle demande alors une barrière de synchronisation globale pour vider les environnements de communication.
- La fonction *store* se charge du stockage des valeurs dans l'environnement de communication.

5.2 Le module Mspml

L'implantation du noyau de la bibliothèque MSPML suit le style de programmation SPMD. Par conséquent, pour un processeur donné, le type des vecteurs parallèles est définis par : **type** α **par** = α . L'utilisation d'un type abstrait pareille permet d'éviter le problème de la programmation data-parallèle SPMD qui ne distingue pas entre les variables qui dépendent de l'identifiant du processeur et celle qui ne dépendent pas de lui.

Ainsi, il est facile d'implanter les primitives de calcul asynchrone :

let mkpar f = f (pid())
let apply f v = f v

Le mécanisme de communication décrit dans la section 4.2.2 est implanté pour la fonction **get**.

L'implantation de la fonction mget suit les mêmes principes mais elle utilise les threads. À un processeur donné i, la liste des fonctions (le premier argument de mget) est stockée dans l'environnement de communication. Alors la fonction $int- > bool$ (le deuxième argument) est appliquée aux entiers entre 0 et $p-1$. À chaque fois que le résultat est **true**, un nouveau thread est crée pour demander la valeur du processeur donné. Quand toutes les requêtes sont satisfaites, le résultat de la fonction est crée. Quand le processeur i reçoit une requête à partir d'un processeur j pour une valeur stockée par mget, si la m-étape du processeur i est égale ou dépasse celle du processeur j, alors la valeur stockée dans l'environnement de communication de i est délinéarisation et appliquée à j pour lui délivrer la valeur v. Cette valeur est linéarisée et envoyée au processeur j.

5.2.1 La composition parallèle

L'implantation de la primitive de composition parallèle est donnée par :

```
let juxta m e1 e2 =
if ((m <= 0) || (m >= (Tcpip.p()))) then failwith "bad parameter in the juxtaposition" else
   begin
     let v =
        begin
        Tcpip.inc_juxta (Tcpip.step);
        Stack.push Tcpip.step.num_mstep Tcpip.env_mstep;
        let pid = Stack.top (Tcpip.env_pid) in
          if (pid < m)
          then
            begin
                Tcpip.new_mstep Tcpip.step m;
                Stack.push m Tcpip.env_p;
                Stack.push (Tcpip.pid()) Tcpip.env_pid;
            e1()
            end
          else
            begin
                Tcpip.new_mstep Tcpip.step ((Tcpip.p())- m);
                Stack.push (Tcpip.p()- m) Tcpip.env_p;
                Stack.push (Tcpip.pid() - m) Tcpip.env_pid;
            e2()
            end
         end
         in
         begin
                Stack.pop Tcpip.env_p;
                Stack.pop Tcpip.env_pid;
                let e = Stack.top (Tcpip.env_mstep) in
                begin
                Tcpip.step.Tcpip.num_mstep.Tcpip.int_mstep <- e.Tcpip.num_mstep.Tcpip.int_mstep;
                Tcpip.step.Tcpip.num_mstep.Tcpip.juxta_mstep <- e.Tcpip.num_mstep.Tcpip.juxta_mstep;
                end;
                Stack.pop Tcpip.env_mstep;
```

```
    v
  end
end
```

l'implantation de la juxtaposition suit le mécanisme de décomposition de la machine décrit dans la section 4.2.2

5.3 Exemple

Soit l'exemple suivant :

```
open Mspml
open Mspmlbase

let _ =
begin
print_string "Exemple 1\n";flush stdout;
initialize();
let e =
let this = mkpar (fun pid → pid) in
let e1 _ = get this this in
let e2 _ = get (mkpar (fun i → i +1)) (mkpar (fun i → i − 1)) in
juxta 2 e1 e2 in
parprint print_int e;
finalize()
end
```

L'exécution parallèle de ce programme avec 4 processeurs dans la machine absolue donne le résultat suivant :

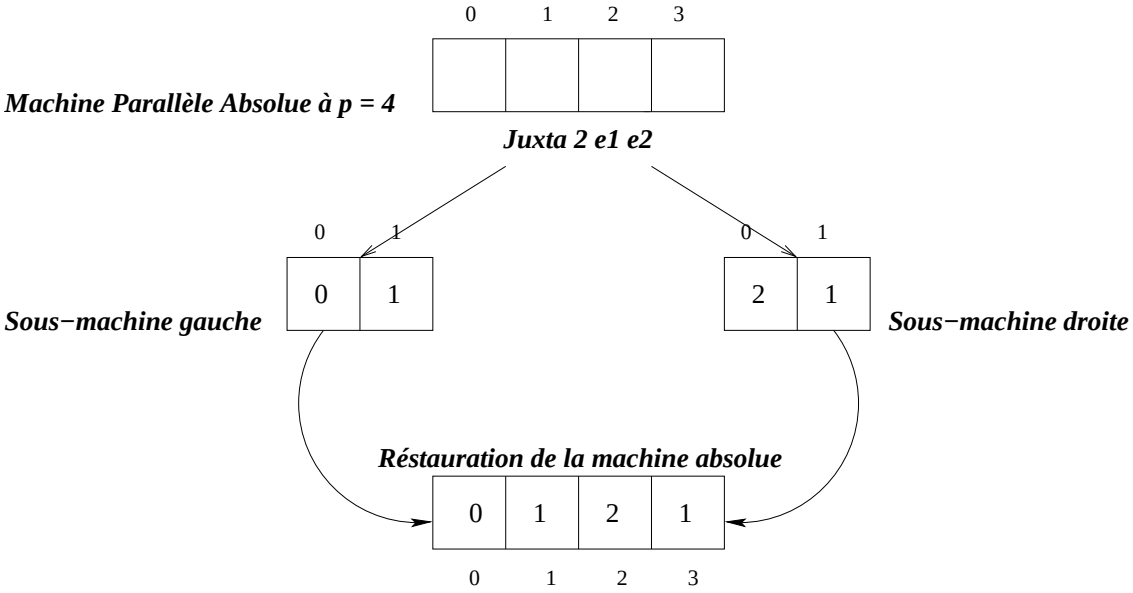

FIG. 5.2 – Exemple de juxtaposition parallèle

Chapitre 6

Conclusion

Les travaux présentés dans ce mémoire de stage d'initiation à la recherche s'inscrivent dans le cadre du projet PROPAC : Programmation Parallèle Certifiée.

Il s'agissait au début d'ajouter l'opération de composition parallèle au langage parallèle fonctionnel Minimally Synchronous Parallel (MSPML) en concevant les sémantiques nécessaires et en ajoutant son implantation à celle de MSPML.

Ainsi, on a d'abord conçu une nouvelle sémantique à grands pas pour MSPML étendant celle présentée dans [20]. Cette nouvelle sémantique contient des évaluations locales et d'autres globales. Ces dernières utilisent un environnement qui contient les valeurs liées à des variables globales, afin d'éviter les substitutions.

Une sémantique à petits pas avec juxtaposition a été aussi conçue. Contrairement à la sémantique à grands pas, celle-ci donne tout les pas de calcul. Dans cette sémantique, le fait d'ajouter l'opération de composition parallèle mène le nombre de processus à changer durant l'exécution d'un programme MSPML. Pour gérer cette contrainte et afin d'être toujours capable de restaurer la machine absolue dès la fin de la composition parallèle, on a introduit des environnements qui stockent les paramètres des processus durant l'exécution.

La confluence de chacune des deux sémantiques (à grands pas et à petits pas) a été prouvée.

Ensuite, on a implanté l'opération de composition parallèle dans MSPML. Pour ce faire, on a utilisé des structures de données de type pile pour les environnements des variables P, Pid et $Mstep$ et une table hachage pour implanter les environnements de communication.

Pour des travaux futurs, il faudra prouver la correction de la sémantique distribuée par rapport à la sémantique à grands pas. Il faudra aussi prouver la sûreté de typage utilisé dans les deux sémantique. Il faudra encore envisager d'adapter notre mécanisme de gestion des environnements de communication à celui proposé dans [4] pour éliminer les barrières de synchronisation.

A la fin de ce stage qui a été très enrichissant pour moi, j'ai eu l'opportunité, d'une part, d'approfondir et de mettre en oeuvre des connaissances déjà acquises durant la partie théorique du Master de recherche M2 IPVGCA, comme par exemple,

les aspects du parallélisme et les systèmes formels, et d'autre part, d'acquérir des compétences nouvelles telle que la programmation fonctionnelle parallèle.

Annexe A

Annexe : preuves des lemmes et propositions

L'évaluation des expressions peut être définie comme une suite de transformations conduisant d'une expression à sa valeur. Ces transformations sont appelées des **réécritures** et une suite de réécriture est appelée **réduction** ou encore **calcul**. Par exemple, si nous nous intéressons à de pures expressions arithmétiques, les réécritures consistent à remplacer une sous-expression constituée d'un opérateur appliqué à deux constantes numériques par le résultat de cette application.

$$(2 + 3) * (4 + 5) \ \rightarrow (2 + 3) * (9) \ \rightarrow 5 * 9 \ \rightarrow 45$$

Une certaine liberté existe dans le choix de la sous-expression qu'on choisit de réduire. Ici on aurait pu aussi procéder de gauche à droite :

$$(2 + 3) * (4 + 5) \ \rightarrow (5) * (4 + 5) \ \rightarrow 5 * 9 \ \rightarrow 45$$

Il est clair que de tels calculs possèdent les deux propriétés suivantes :

- **Finitude :** tous les calculs se terminent.
- **Cohérence :** tous les calculs aboutissent au même résultat.

La propriété de cohérence résulte d'une propriété plus générale que l'on appelle **confluence**. Un ensemble de règles de réécriture est dit **confluent** si dans toute situation où une même expression e peut se réécrire en deux expressions e_1 et e_2, il existe une expression e_3 telle que e_1 et e_2 se réécrivent en e_3. Cette propriété est illustrée graphiquement su la figureA et si on représente graphiquement l'ensemble des réécritures possibles de l'expression $(2 + 3) * (4 + 5)$ (voir la figure A), cette propriété apparaît immédiatement sur le dessin.

Lorsqu'on passe des expressions arithmétiques à un langage de programmation permettant l'écriture de fonctions récursives, la propriété de finitude de tous les calculs ne peut pas être conservée. Par contre, la propriété de confluence n'entraîne pas la cohérence au sens utilisé plus haut (tous les calculs aboutissent au même résultat) mais une forme de cohérence plus faible[1].

[1] **cohérence faible :** tous les calculs qui se terminent aboutissent au même résultat.

En effet, dans des systèmes de réécriture admettant des calculs infinis, il peut exister des expressions possédant à la fois des calculs finis et des calculs infinis. Ces systèmes ne peuvent être que **faiblement cohérents**.

Cette propriété de cohérence faible ne vaut que pour les langages purement fonctionnels et elle est essentielle pour la vérification de programmes[14]

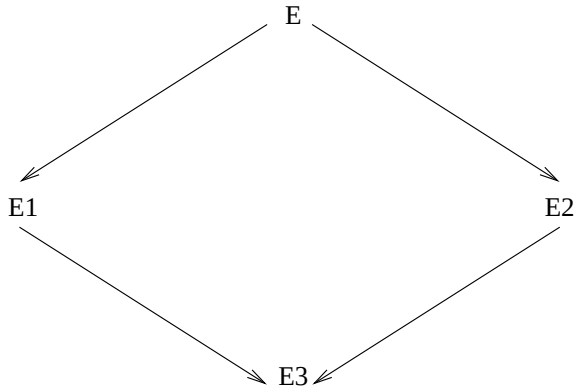

FIG. A.1 – La propriété de confluence

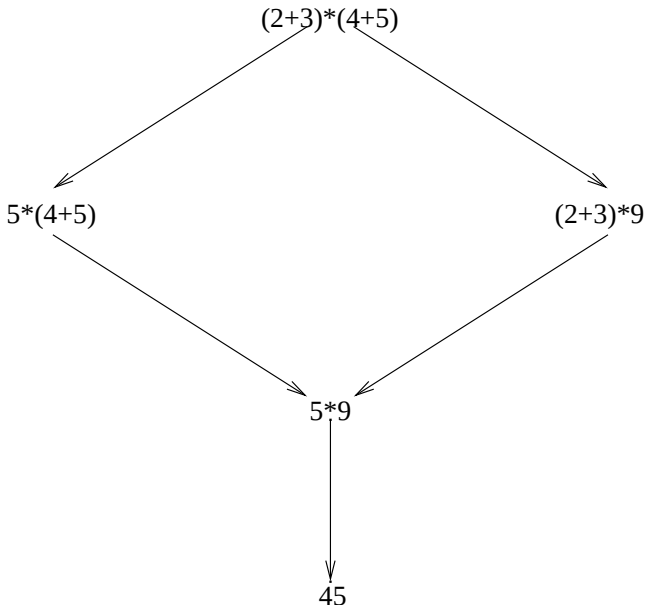

FIG. A.2 – Un graphe de réécriture

A.1 Confluence de la sémantique à grand pas

A.1.1 Preuve de la proposition 2

On présente ici la preuve de la proposition 2 donné au chapitre 3.

Par induction sur la structure de dérivation $e \triangleright_{loc} v_1$. La preuve est faite par énumération des règles appliquées à la racine de la dérivation $e \triangleright_{loc} v_1$.

1. $e \triangleright_{loc} v_1$ est (3.35). Elle est la seule règle applicable à la racine.
2. $e \triangleright_{loc} v_1$ est (3.36). Elle est la seule règle applicable à la racine.
3. $e \triangleright_{loc} v_1$ est (3.39). Elle est la seule règle applicable à la racine.
4. $e \triangleright_{loc} v_1$ est (3.40). Elle est la seule règle applicable à la racine.
5. $e \triangleright_{loc} v_1$ est (3.41). Elle est la seule règle applicable à la racine.
6. $e \triangleright_{loc} v_1$ est (3.44). Ellee est la seule règle applicable à la racine.
7. $e \triangleright_{loc} v_1$ est (3.45). Elle est la seule règle applicable à la racine.
8. $e \triangleright_{loc} v_1$ est (3.37). Les règles (3.37) et (3.38) sont applicables à la racine.
   a. si $e \triangleright_{loc} v_2$ est la règle (3.37), alors $v_1 = v_2$.
   b. il est impossible d'appliquer la règle (3.38) car dans cette règle $e_1 \triangleright_{loc} \mathbf{false}$, or, dans (3.37) on a $e_1 \triangleright_{loc} \mathbf{true}$.
9. $e \triangleright_{loc} v_1$ est (3.38). Les règles (3.38) et (3.37) sont applicables à la racine.
   a. si $e \triangleright_{loc} v_2$ est la règle (3.38), alors $v_1 = v_2$.
   b. il est impossible d'appliquer la règle (3.37) car dans cette règle $e_1 \triangleright_{loc} \mathbf{true}$, or, dans (3.38) on a $e_1 \triangleright_{loc} \mathbf{false}$.
10. $e \triangleright_{loc} v_1$ est (3.42). Les règles (3.43), (3.42) et (3.46) sont applicables à la racine.
    a. si $e \triangleright_{loc} v_2$ est la règle (3.42), alors $v_1 = v_2$.
    b. il est impossible d'appliquer la règle (3.43) car dans cette règle $e_1 \triangleright_{loc} +$, or, dans (3.42) on a $e_1 \triangleright_{loc} \mathbf{fix}$.
    c. il est impossible d'appliquer la règle (3.46) car dans cette règle $e_2 \triangleright_{loc} \mathbf{fun}\ x : \mathcal{L} \rightarrow e$, or, dans (3.42) on a $e_2 \triangleright_{loc} \mathbf{op}$.
11. $e \triangleright_{loc} v_1$ est (3.43). Les règles (3.43), (3.42) et (3.46) sont applicables à la racine.
    a. si $e \triangleright_{loc} v_2$ est La règle (3.43), alors $v_1 = v_2$.
    b. il est impossible d'appliquer la règle (3.42) car dans cette règle $e_1 \triangleright_{loc} \mathbf{fix}$, or, dans (3.43) on a $e_1 \triangleright_{loc} +$.
    c. il est impossible d'appliquer la règle (3.46) car dans cette règle $e_1 \triangleright_{loc} \mathbf{fix}$, or, dans (3.43) on a $e_1 \triangleright_{loc} +$.
12. $e \triangleright_{loc} v_1$ est (3.46). Les règles (3.46), (3.42) et (3.43) sont applicables à la racine.
    a. si $e \triangleright_{loc} v_2$ est La règle (3.46), alors $v_1 = v_2$.

b. il est impossible d'appliquer la règle (3.42) car dans cette règle $e_2 \triangleright_{loc}$ **op**, or, dans (3.46) on a $e_2 \triangleright_{loc}$ **fun** $x : \mathcal{L} \rightarrow e$.

c. il est impossible d'appliquer la règle (3.43) car dans cette règle $e_1 \triangleright_{loc} +$, or, dans (3.46) on a $e_1 \triangleright_{loc}$ **fix**.

Donc, la réduction locale de la sémantique à grand pas avec juxtaposition est confluente.

A.1.2 Preuve de la proposition 3

On présente ici la preuve de la proposition 3 donné au chapitre 3.

Par induction sur la structure de dérivation $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$. La preuve est faite par énumération des règles appliquées à la racine de la dérivation $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$.

1. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.47). Elle est la seule règle applicable à la racine.
2. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.48). Les règles (3.48) et (3.49) sont applicables à la racine.
   a. si $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est la règle (3.48), alors $v_1 = v_2$.
   b. il est impossible d'appliquer la règle (3.49) car la variable est lié à une fermeture, or, dans (3.48) elle est liée à un vecteur.
3. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.49). Les règles (3.49) et (3.48) sont applicables à la racine.
   a. si $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est la règle (3.49) , alors $v_1 = v_2$.
   b. il est impossible d'appliquer la règle (3.48) car la variable est lié à une fermeture, or, dans (3.49) elle est liée à un vecteur.
4. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.50). Les règles (3.50) et (3.51) sont applicables à la racine.
   a. si $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est la règle (3.50) , alors $v_1 = v_2$.
   b. il est impossible d'appliquer la règle (3.51) car $(e_2 : \mathcal{L})$ local, or, dans (3.50) elle est de type global $(e_2 : \mathcal{G})$.
5. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.51). Les règles (3.51) et (3.50) sont applicables à la racine.
   a. si $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est la règle (3.51) , alors $v_1 = v_2$.
   b. il est impossible d'appliquer la règle (3.51) car $(e_2 : \mathcal{G})$ global, or, dans (3.50) elle est de type local $(e_2 : \mathcal{L})$.
6. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.52). Elle est la seule règle applicable à la racine.
7. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.56). Elle est la seule règle applicable à la racine.
8. $\mathcal{E} \vdash_f^p e_1 \triangleright_{\text{glob}} v_1$ est (3.57). Elle est la seule règle applicable à la racine.

9. $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est (3.58). Les règles (3.58) et (3.59) sont applicables à la racine.
   - **a.** si $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est la règle (3.58) , alors $v_1 = v_2$.
   - **b.** il est impossible d'appliquer la règle (3.59) car la valeur au processeur est **true** ($v_n \equiv$ **true**) , or, dans (3.59) elle est **false**, ce qui est impossible.
10. $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est (3.59). Les règles (3.59) et (3.58) sont applicables à la racine.
    - **a.** si $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est la règle (3.59) , alors $v_1 = v_2$.
    - **b.** il est impossible d'appliquer la règle (3.58) car la valeur au processeur est **false** ($v_n \equiv$ **false**) , or, dans (3.59) elle est **true**, ce qui est impossible.
11. $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est (3.60). la règle (3.60) est l'unique règle applicable.
12. $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est (3.53). Les règles (3.53), (3.54) et (3.55 sont applicables à la racine.
    - **a.** si $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est La règle (3.53) , alors $v_1 = v_2$.
    - **b.** il est impossible d'appliquer la règle (3.54) car $(e_2 : \mathcal{L})$ global, or, dans (3.53) elle est de type local $(e_2 : \mathcal{G})$.
    - **b.** il est impossible d'appliquer la règle (3.55) car $\mathcal{E}_g \vdash_f^{p'} e_1 \rhd_{\text{glob}} [(\textbf{fun}\ x : \mathcal{G} \rightarrow e), \mathcal{E}']$, or, dans (3.53) $(e_1 : \mathcal{L})$ et $e_1 \rhd_{loc}$ **fix**.
13. $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est (3.54). Les règles (3.53), (3.54) et (3.55 sont applicables à la racine.
    - **a.** si $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est La règle (3.54) , alors $v_1 = v_2$.
    - **b.** il est impossible d'appliquer les règles (3.53)(3.55) car $(e_2 : \mathcal{G})$ global, or, dans (3.54) elle est de type local $(e_2 : \mathcal{L})$.
14. $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est (3.55). Les règles (3.53), (3.54) et (3.55 sont applicables à la racine.
    - **a.** si $\mathcal{E} \vdash_f^p e_1 \rhd_{\text{glob}} v_1$ est La règle (3.55) , alors $v_1 = v_2$.
    - **b.** il est impossible d'appliquer les règles (3.53)(3.54) car dans la règle (3.55) $(e_1 : \mathcal{L})$ et $e_1 \rhd_{loc}$ **fix**.

Donc, la réduction globale de la sémantique à grand pas avec juxtaposition est confluente.

A.2 Confluence de la sémantique à petit pas

A.2.1 Preuve du lemme 1

On présente ici la preuve de la proposition 1 donné au chapitre 4.

Par induction sur la structure de dérivation $(e, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{mstep}) \rightharpoonup (v_1, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})$. La preuve est faite par énumération des règles

appliquées à la racine de la dérivation
$(e, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (v_1, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})$.

Afin de simplifier la présentation de la preuve nous renommant :
$(e, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (v_1, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})$ en (I)
et
$(e, step, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step}) \rightharpoonup (v_2, step_1, abs\_pid_i, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}, \mathcal{E}_{step})$ en (II).

1. Dans le cas où les arguments d'une expression donnée ne sont pas des valeurs, alors dans ce cas on applique la règle (4.7) avec une des autres règles locales.
   (a) L'utilisation du contexte vide [] est possible et dans ce cas là, la règle du contexte est omis et on ne considère qu'une des autres règles locale.
   (b) Il est aussi possible d'avoir deux contextes différents pour le même terme. Dans ce cas là on a :
      i. soit l'application de la règle 4.7 sur un terme va donné un contexte qui est équivalent au deuxième.
      ii. soit on applique la règle de contexte sur les deux et qui donne des termes différents, là on applique les autres règles locales(confluence des autres règles est prouvée ci-dessous).
2. (I) est (4.8). Elle est la seule règle applicable à la racine.
3. (I) est (4.9). Elle est la seule règle applicable à la racine.
4. (I) est (4.10). Elle est la seule règle applicable à la racine.
5. (I) est (4.11). Elle est la seule règle applicable à la racine.
6. (I) est (4.12). Elle est la seule règle applicable à la racine.
7. (I) est (4.13). Elle est la seule règle applicable à la racine.
8. (I) est (4.14). Elle est la seule règle applicable à la racine.
9. (I) est (4.15). Elle est la seule règle applicable à la racine.
10. (I) est (4.16). Elle est la seule règle applicable à la racine.
11. (I) est (4.17). Elle est la seule règle applicable à la racine.
12. (I) est (4.18). Elle est la seule règle applicable à la racine.
13. (I) est (4.19). Les règles applicables à la racine sont (4.19) et (4.20).
    a. (4.19) est une règle applicable dans le cas où le processus qui l'exécute fait une communication avec un autre.
    b. (4.20) est une règle applicable dans la cas où le processus fait une communication avec lui même.
14. (I) est (4.21).Les règles applicables à la racine sont (4.21) et (4.22).
    a. (4.21) est une règle applicable dans le cas où le processus demandant est celui répondant.
    b. (4.22) est une règle applicable dans la cas où le processus fait une communication avec lui même.

15. (I) est (4.23). Les règles applicables à la racine sont (4.23) et (4.24).
    a. (4.23) est une règle applicable que pour les processus dont le $pid < m$ tel que m est le premier argument de la juxtaposition. Dans ce cas c'est la seule règle applicable.
    b. (4.24) est une règle applicable pour les processus dont le $pid > m - 1$ tel que m est le premier argument de la juxtaposition. Dans ce cas c'est la seule règle applicable.
16. (I) est (4.23). Même que le cas 16.
17. (I) est (4.25).C'est la seule règle applicable à la racine.

La réduction locale est confluente.

A.2.2 Preuve de la proposition 5

Le déterminisme dû à l'utilisation de contexte dans la réduction locale n'est pas présent ici, d'où la proposition 5

On présente ici la preuve de la proposition 5 donné au chapitre 4.

Dans la réduction globale on a deux possibilités :

soit, $\ll (e_{1/0}, mstep_0, abs\_pid_0, \mathcal{E}_{c0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{mstep_0}) , \ldots ,$
$(e_{2/i}, mstep_i, abs\_pid_i, \mathcal{E}_{ci}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}, \mathcal{E}_{mstep_i})$
$, \ldots , (e_{1/(p-1)}, mstep_{(p-1)}, abs\_pid_{(p-1)}, \mathcal{E}_{c(p-1)}, \mathcal{E}_{p_{(p-1)}}, \mathcal{E}_{pid_{(p-1)}}, \mathcal{E}_{mstep_{(p-1)}}) \gg$

ou, $\ll (e_{1/0}, mstep_0, abs\_pid_0, \mathcal{E}_{c0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}, \mathcal{E}_{mstep_0}) , \ldots ,$
$(e_{3/j}, mstep_j, abs\_pid_j, \mathcal{E}_{cj}, \mathcal{E}_{p_j}, \mathcal{E}_{pid_j}, \mathcal{E}_{mstep_j})$
$, \ldots , (e_{1/(p-1)}, mstep_{(p-1)}, abs\_pid_{(p-1)}, \mathcal{E}_{c(p-1)}, \mathcal{E}_{p_{(p-1)}}, \mathcal{E}_{pid_{(p-1)}}, \mathcal{E}_{mstep_{(p-1)}}) \gg$

1. si $i = j$ et une des deux règles appliquée est locale, dans ce cas l'autre règle est forcement locale. Par la confluence locale prouvée ci-dessus on a $e_2 = e_3$.
2. si $i = j$ et une des règle est **request**, la seule règle applicable est (4.27). Le résultat qu'elle donne est unique donc on a $e_2 = e_3$.
3. si $i \neq j$ et les deux règles appliquées sont locales alors on obtient :

   $\ll e_1/0, ..., e_2/i, ..., e_1/p-1 \gg$ dans un cas et $\ll e_1/0, ..., e_2/j, ..., e_1/p-1 \gg$ dans l'autre. Il suffit alors d'appliquer la règle locale (et la règle (4.26) utilisée dans le premier cas sur le e1/i de la seconde expression distribuée et la règle locale (et la règle 4.20) utilisée dans le second cas sur le e1/j de la première expression distribuée. On obtient alors la même expression distribuée : $\ll e_0/i, ..., e_2/i, ..., e_2/j, ..., e_1/p-1 \gg$
4. si $i \neq j$ et les deux règles appliquées sont (4.27) (c-à-d avoir des **request** au niveau de e_1/i et e_1/j) alors :

(a) soit la requête de communication du processus i est déstinée à un processus i' et celle de j à un processus j' tel que $i' \neq j$ et $j' \neq i$ dans ce cas là il y a aucun problème la règle (4.27) deux fois est on obtient la même expression distribuée : $\ll e_0/i, ..., e_2/i, ..., e_2/j, ..., e_1/p-1 \gg$.

(b) soit la requête est mutuelle entre les deux processus i et j, c-à-d que $i' = j$ et ou $j' = i$ alors le résultat de l'application de la règle 4.27) qui peut être appliquée dans les deux sens ne cause pas de problèmes, car cette règle ne modifie que le premier composant du tuple.

5. si $i \neq j$ et une des deux règles applicables est une règle locale (au niveau de e_1/i) et l'autre est un request , alors :

   (a) soit le request demande une valeur à un processus j' tel que $j' \neq i$ dans ce cas il n y a auncun problème dans l'ordre d'application.

   (b) soit le request demande une valeur à un processus j' tel que $j' = i$ dans ce cas on a deux possibilitées :

      i. soit on applique la règle locale et puis le request (4.27).

      ii. soit on applique la règle (4.27) puis la règle locale.

      Dans les deux cas le même résultat est obtenu même dans le cas où la règle locale change l'environnement de communication. Parce que si le processus j demande une valeur au processus i à une m-étape $step_i$ donnée, on à deux cas :

      i. soit i et j ont atteint la même m-étape. Dans ce cas là aucun problème ne se pose dans les deux cas d'application et le même résultat est obtenu.

      ii. soit i et j n'ont pas atteint la même m-étape est dans ce cas là l'application de la règle (4.27) en premier ou pas ne change rien puisque, dans ce cas si la m-étape de i est inférieur à celle de j à la m-étape $step_i$ de la requête du processus j. Le procesus j doit attendre le processus i qu'il fasse sa règle locale afin d'atteindre la m-étape $step_i$ et la on revient implicitement au même ordre d'application : règle locale puis règle globale.

Pour le premier cas, l'application dans un sens ou l'autre ne change rien

Bibliographie

[1] K. Apt and R.E Olderog. *Verification of Sequential and Concurrent Programs*. Springer Verlag, 1997.

[2] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3) :317–328, 2003.

[3] M. Bamha and G. Hains. Frequency-adaptive join for Shared Nothing machines. *Parallel and Distributed Computing Practices*, 2(3) :333–345, 1999.

[4] A. Belbekkouche. Mspml : Environnements de communication et tolérance aux pannes. Mémoire de master de recherche en informatique, LIFO, Université d'Orléans, Septembre 2005.

[5] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[6] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations : Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[7] V. Blanco, J. A. González, C. León, C. Rodríguez, G. Rodríguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30 :337–356, 2004.

[8] L. Bougé. Le modèle de programmation à parallélisme de données : une perspective sémantique. *RAIRO Technique et Science Informatiques*, 12(5), 1993.

[9] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. Formal Validation of Data–Parallel Programs : a Two–Component Assertional Proof System for a Simple Language. *Theoretical Computer Science*, 189(1-2) :71–107, 1997.

[10] L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Euro-Par'96 Parallel Processing*, number 1123–1124 in Lecture Notes in Computer Science, Lyon, August 1996. LIP-ENSL, Springer.

[11] A. Braud and C. Vrain. Parallélisation d'algorithmes génétiques fondée sur le modèle BSP. In Michèle Sebag, editor, *Première Conférence d'Apprentissage (CAP'99)*, juin 1999. To appear.

[12] R. Calinescu. Bulk synchronous parallel scheduling of uniform dags. In Bougé et al. [10].

[13] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[14] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Ediscience International, 1995.

[15] D. C. Dracopoulos and S. Kent. Speeding up genetic programming : A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.

[16] F. Gava and F Loulergue. A Functional Language for Departmental Metacomputing. Technical Report 2004-09, University of Paris 12, LACL, 2004.

[17] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.

[18] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In Bougé et al. [10].

[19] Y. Kee and S. Ha. An Efficient Implementation of the BSP Programming Library for VIA. *Parallel Processing Letters*, 12(1) :65–77, 2002.

[20] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3) :182–199, 2004. W. Dosch, editor, special issue on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing.

[21] Frédéric Loulergue. *Programmation fonctionnelle d'ordinateurs parallèles et de méta-ordinateurs : sémantiques, systèmes et preuves*. Mémoire d'habilitation à diriger des recherches, University Paris Val de Marne, décembre 2004.

[22] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency : Practice and Experience*, 11(9) :461–477, 1999.

[23] C. Rodriguez, J.L. Roda, F. Sande, D.G. Morales, and F. Almeida. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 26 :753–767, 2000.

[24] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6) :409–424, 1998.

[25] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

[26] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[27] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990.

[28] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28 :1709–1732, 2002.

[29] J. Garrigue D. Rémy X. Leroy, D. Doligez and J. Vouillon. The Objective Caml System 3.08, Documentation and User's Guide. Technical report, 2004. `www.ocaml.org`.