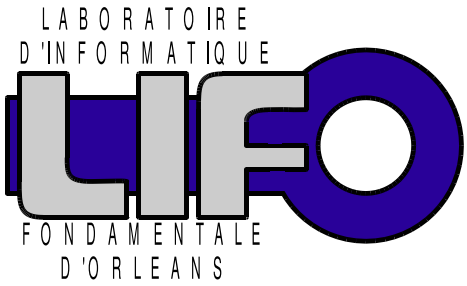

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Syntaxe et sémantique de Revised Bulk Synchronous Parallel ML

Wadoud Bousdira
Frédéric Loulergue
Louis Gesbert

Rapport n^o **RR-2010-12**

Syntaxe et sémantique de Revised Bulk Synchronous Parallel ML

Wadoud Bousdira
LIFO, University of Orléans, France
Wadoud.Bousdira@univ-orleans.fr

Frédéric Loulergue
LIFO, Université d'Orléans, France
Frederic.Loulergue@univ-orleans.fr

Louis Gesbert
MLstate, Paris, France,
Louis.Gesbert@mlstate.com

27 octobre 2010

Résumé

Bulk Synchronous Parallel ML (BSML) est une extension du langage fonctionnel Objective Caml, fondé sur un modèle structuré de parallélisme, le modèle BSP. Ce modèle assure au programmeur BSML la sûreté d'exécution tout en lui laissant le strict contrôle des processeurs. Le modèle de prévision de performances de BSML est simple et réaliste. Le parallélisme est exprimé en utilisant un ensemble de primitives fonctionnelles pures sur une structure de données parallèle appelée vecteur parallèle. Cependant, les programmes sont souvent difficiles à écrire et leur mise au point peut être fastidieuse. Nous proposons dans cet article une nouvelle syntaxe et une sémantique associée dans le but d'écrire des programmes plus courts et plus lisibles. Nous formalisons la syntaxe et la sémantique classiques ainsi que les nouvelles syntaxe et sémantique, puis nous les modélisons en Coq. Leur confluence est établie.

Abstract

Bulk Synchronous Parallel ML (BSML) is an extension of the functional language Objective Caml to program Bulk Synchronous Parallel (BSP) algorithms. BSML programs are deterministic and deadlock free and have predictable performances. Parallelism is expressed with a set of four pure functional primitives on a parallel data structure called parallel vector. However, the programs are often hard to read and so to debug. We propose a new syntax and the associated semantics which allow to write shorter and more readable programs. The formalisation of the classical and new semantics as well as the proofs of confluence are established using the Coq proof assistant.

Table des matières

1 Introduction

Bulk Synchronous Parallel ML ou BSML permet une approche simple et claire du parallélisme. Il se base sur le langage de programmation fonctionnelle Objective Caml [2] et grâce à une extension par des primitives parallèles, permet d'écrire des programmes BSP (*Bulk Synchronous Parallel*, soit parallélisme quasi-synchrone).

Le modèle BSP [28] décrit une machine parallèle comme un ensemble de paires processeur-mémoire homogènes en nombre fixe. Les possibilités d'échange entre les processeurs sont rendues possibles par un réseau de communication et une unité de synchronisation globale. La particularité du modèle BSP est que l'échange de données entre processeurs n'est effectif qu'une fois une barrière de synchronisation globale effectuée. Ainsi, on a l'assurance que toutes les communications se sont terminées et les problèmes de concurrence sont très restreints. L'exécution est ainsi découpée en super-étapes chacune composée de (a) une phase de calculs locaux, indépendants, sur chaque processeur, (b) une phase de communication procédant à tous les échanges demandés entre processeurs, et (c) une phase de synchronisation. Ce découpage garantit que l'exécution est déterministe et assure l'absence d'inter-blocage.

Le modèle BSP fournit également un modèle de coûts, simple et réaliste, qui facilite le raisonnement sur les performances des algorithmes et programmes BSP. Le modèle de coût est paramétré par quatre valeurs caractéristiques des architectures parallèles dont le nombre p de processeurs. Nous renvoyons à [27] par exemple pour plus de détails sur le modèle de coûts BSP. En pratique, le modèle BSP a été largement utilisé dans des domaines divers tels que l'intelligence artificielle [26, 9], les algorithmes pour les bases de données [6], le calcul numérique [25, 7], *etc.*

BSML est actuellement implanté comme une bibliothèque pour le langage Objective Caml. Il se base sur une extension confluente du λ-calcul et il est possible de certifier la correction de programmes BSML [17] à l'aide de l'assistant de preuve Coq [1]. La bibliothèque BSML a été développée depuis quelques années et des extensions dans plusieurs directions sont implantées. La dernière version de BSML est disponible à l'adresse suivante :

`http://traclifo.univ-orleans.fr/BSML`

Un programme BSML est conçu comme un programme séquentiel s'exécutant sur un vecteur parallèle, qui à lui seul permet l'accès au parallélisme. Un vecteur parallèle a pour type α **par** et contient p valeurs locales de type α. Chacune de ces valeurs est hébergée dans la mémoire de l'un des processeurs de la machine parallèle. Le nombre de processeurs p est défini par une constante **bsp_p** tout au long de l'exécution du programme. Les vecteurs parallèles sont manipulés en utilisant 4 primitives qui constituent le noyau de BSML. Deux d'entre elles sont asynchrones et correspondent à un accès local aux valeurs de chaque composante des vecteurs. Les deux autres effectuent la communication et sont synchrones. BSML est complètement formalisé avec ces seules primitives. Cependant, le non emboîtement des vecteurs parallèles et l'obligation même de ne pas évaluer d'expression parallèle dans le contexte d'un vecteur parallèle (ce qui peut être imposé par un système de types [19, 20]), peut rendre la manipulation des vecteurs parallèles fastidieuse.

C'est pourquoi nous proposons une syntaxe révisée pour BSML, plus simple que la syntaxe classique et qui vise à faciliter la manipulation des vecteurs parallèles.

Dans un premier temps, nous présentons informellement la programmation en BSML classique (section 2) et en BSML révisé (section 3). Puis, nous définissons μBSML (section 4) et $\mu\text{BSML}^{\$}$ (section 5) des sémantiques formelles de noyaux de BSML classique et de BSML révisé respectivement. Nous modélisons ensuite ces sémantiques en Coq et établissons leur confluence (section 6). La section 7 présente des travaux connexes et la section 8 est consacrée à la conclusion et aux perspectives de ce travail.

2 Programmation parallèle avec BSML

Pour intégrer le parallélisme, BSML se base sur une structure de données qui est le vecteur parallèle. Un vecteur parallèle est de type α **par** et contient p valeurs de type α. Le nombre de processeurs p est noté **bsp_p**. Les p processeurs de la machine BSP sont identifiés par des entiers de 0 à $p-1$ appelés *pid* (pour "*Processor IDentifier*") des processeurs. Un vecteur parallèle est décrit par la notation suivante :

$$\langle x_0, x_1, \ldots, x_{p-1} \rangle : \alpha\,\texttt{par}$$

ou en notation abrégée $\langle x_i \rangle_i$. Ce vecteur contient la valeur x_i au processeur i, avec tous les x_i de type α. On note dans le cas général $\langle\ \rangle_i$ pour signifier que le pid du processeur est représenté par l'index i dans le vecteur, $\langle i \rangle_i$ étant donc le vecteur des pids. On ne peut accéder à une valeur locale x_i que dans deux cas :

1. lors de calculs locaux sur le processeur i ou,
2. après des communications.

Ces restrictions sont inhérentes au parallélisme à mémoire répartie. L'usage du type opaque α les garantit. Le parallélisme est rendu explicite et les programmes sont plus lisibles. Étant donné que BSML gère à la fois une machine parallèle dans son ensemble et des processeurs individuels, une distinction entre les différents niveaux d'exécution à l'intérieur de la machine parallèle devient nécessaire.

L'exécution *répliquée* est celle par défaut. Le code BSML est séquentiel et ne contient aucune primitive parallèle. Ce code est exécuté comme si la machine disposait d'un processeur unique. Dans la pratique, son exécution est simultanée sur tous les processeurs, garantissant ainsi d'obtenir le même résultat partout.

L'exécution *locale* est celle qui se produit dans les vecteurs parallèles, sur chacune de leurs composantes : les processeurs utilisent leurs données locales pour faire des calculs qui peuvent différer de l'un à l'autre.

L'exécution *globale* concerne les processeurs dans leur ensemble en tant que tout. Typiquement, on considère les communications comme une opération globale.

Le tableau de la figure 1 récapitule les primitives parallèles BSML :

Primitive	**Type**	**Description**
mkpar	(int → α) → α **par**	$f \mapsto \langle f\,0\ ,\ \ldots\ ,\ f\,(p-1)) \rangle$
apply	(α → β) **par** → α **par** → β **par**	$\langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle v_0, \ldots, v_{p-1} \rangle \mapsto \langle f_0\,v_0, \ldots, f_{p-1}\,v_{p-1} \rangle$
proj	α **par** → int → α	$\langle v_0, \ldots, v_{p-1} \rangle \mapsto$ (**fun** $i \to v_i$)
put	(int → α) **par** → (int → α) **par**	$\langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle$**fun** $i \to f_i 0, \ldots,$ **fun** $i \to f_i(p-1) \rangle$

FIGURE 1 – Les primitives parallèles de BSML classique

mkpar construit un vecteur parallèle dont les composantes sont les résultats de l'application locale de la fonction passée en argument au pid de chaque processeur. Par exemple, pour répliquer une valeur : **let** replicate x = **mkpar** (**fun** _ → x) et nous avons : replicate x = $\langle\ x\ , \ldots,\ x\ , \ldots,\ x\ \rangle$.

apply est la primitive des calculs locaux. Elle applique une fonction locale à un paramètre local sur chaque processeur. Par exemple, si on veut appliquer une même fonction à tous les membres d'un vecteur parallèle : **let** parfun f v = **apply** (replicate f) v.

proj est la primitive duale de **mkpar** et le seul moyen d'extraire une valeur non-parallèle à partir d'un vecteur parallèle. Étant donné un vecteur, **proj** renvoie une fonction non parallèle qui, appliquée à un pid, renvoie la valeur de la composante du vecteur correspondant à ce pid. **proj** est

souvent utilisé à la fin d'un calcul parallèle pour rassembler les résultats obtenus. Par exemple, pour convertir un vecteur parallèle en une liste, on écrit : **let** proj_list v = List.map (**proj** v) procs_list où procs_list est la liste des identifiants de processeurs [0; 1; ... ; **bsp_p**−1]. Il est à noter qu'il ne faut pas qu'une application de **proj** s'exécute dans le contexte d'un vecteur parallèle.

put est la primitive générique de communications : elle permet de décrire tout échange de valeurs locales entre processeurs. Elle met implicitement fin à la super-étape en cours. L'usage canonique de **put** est :

put (**mkpar** (**fun** sender sendto → e)

où l'expression e calcule (ou plutôt sélectionne) en fonction des pids sender et sendto, la valeur qui devra suivre le chemin de l'un à l'autre. La valeur renvoyée par **put** est un autre vecteur de fonctions : au processeur j, la fonction appliquée à i retourne la valeur que le processeur j a reçue du processeur i.

3 Une syntaxe révisée pour BSML

Définir le parallélisme avec un petit nombre d'opérations est une grande force pour la formalisation du langage. De cette façon, les définitions sont claires et les preuves plus simples. Pouvoir intégrer ces opérations dans des fonctions de plus haut niveau permet d'obtenir des traitements parallèles complexes de manière transparente. Malgré tout, les programmes même de haut niveau, doivent gérer les valeurs répliquées et les vecteurs parallèles, et l'usage des primitives ne se révèle pas des plus pratiques. En effet, toute opération à l'intérieur d'un vecteur parallèle nécessite, outre un appel à une primitive, la définition d'une fonction ad-hoc. Lorsqu'on travaille avec plusieurs vecteurs parallèles, les appels à **apply** s'imbriquent et font obstacle à la lisibilité. Par exemple, simplement pour transformer une paire de vecteurs en un vecteur de paires, on écrit

let combine_vectors(v, w) = **apply**(parfun(**fun** v w → v, w) v) w

Ce code pourrait être plus simple avec la définition de

let parfun2 f x y = **apply** (parfun f x) y

On obtient alors :

let combine_vectors(v, w) = parfun2(**fun** v w → v, w) v w

Le code est plus lisible, mais pas encore satisfaisant car il nécessite la définition d'une fonction spécifique. Cela implique de créer des paramètres nommés même si la fonction n'a qu'un cas d'application. Ces noms de paramètres sont redondants et peuvent provoquer des erreurs :

let combine_vectors(v, w) = parfun2 (**fun** w v → w, v) v w

qui équivaut à la fonction ci-dessus mais peut mener à confusion.

Pour simplifier l'écriture du code, plutôt que le point de vue de BSML basé sur les primitives, on considère un point de vue qui s'articule autour de la notion de niveau d'exécution. L'utilisateur peut alors déclarer si son code doit être exécuté globalement comme en Objective Caml standard, ou localement à partir d'un vecteur parallèle. L'avantage de cette approche est de pouvoir accéder aux composantes locales d'autres vecteurs depuis l'intérieur du code local. L'utilisation de fonctions additionnelles devient inutile. Une section locale est représentée par ≪≫, le code entre chevrons devant être exécuté localement. L'information répliquée est disponible à l'intérieur du vecteur et ≪ x ≫ désigne le vecteur contenant x partout. De plus, pour accéder à la composante locale d'un vecteur v, on ajoute la syntaxe v. $ $ peut bien sûr être utilisé uniquement à l'intérieur des sections locales. Il est alors possible d'écrire combine_vectors comme suit :

Notation	Type	Description
$\ll$ e $\gg$	t **par** si e : t	$\langle e, \ldots, e \rangle$
$this$ (dans une section locale)	int	i sur le processeur i
v (dans une section locale)	t (si v : t **par**)	v_i sur le processeur i (si $v = \langle v_0, \ldots, v_{p-1} \rangle$)

FIGURE 2 – Résumé de la syntaxe révisée de BSML

let combine_vectors (v, w) = $\ll$ v, w $\gg$

ce qui s'avère plus court, plus clair et moins sujet à l'erreur. Enfin, on peut accéder au pid local avec $this$, ce qui dispense de l'usage de **mkpar**. Les primitives synchrones (**proj** et **put**) ne nécessitent pas une syntaxe spécifique, mais leur usage devient plus simple.

Les deux syntaxes ont la même expressivité. En effet, $\ll \gg$ combiné avec $this$ est équivalent à **mkpar** avec la définition :

let mkpar f = $\ll$ f $this$ $\gg$

et l'utilisation de tout autre vecteur dans $ $ est équivalente à un appel de **apply**. Par exemple,

let p = **put**(**apply** (**mkpar** (**fun** sendfrom x sendto $\rightarrow$ e (sendfrom, sendto, x))) x)

qui calcule les valeurs à communiquer en fonction de la source, de la destination et d'un vecteur x peut maintenant s'écrire :

let p = **put** $\ll$ **fun** sendto $\rightarrow$ e($this$, sendto, x)$\gg$

La figure 2 récapitule la syntaxe révisée de BSML.

On peut informellement établir une transformation qui modifie le code qui utilise une section locale en code qui utilise les primitives. Cette transformation est relativement simple et elle suit les étapes suivantes :

- le contenu d'une section locale $\ll$ e $\gg$ devient le corps d'une fonction **fun** ... $\rightarrow$ e,
- cette fonction est construite automatiquement de façon récursive, à partir des $ $ de la section locale qui deviennent ses paramètres,
- **mkpar** est appliqué à cette fonction, dont le premier argument correspond à $this$,
- des appels appropriés à **apply** sont alors déroulés autour de cet appel correspondant à chacun des vecteurs parallèles utilisés.

4 μBSML : sémantique classique

Nous considérons dans cette section un sous-ensemble de BSML suffisamment simple mais qui englobe toutes les fonctionnalités importantes de BSML. Il inclut les opérations parallèles et un noyau purement fonctionnel de ML que nous appelons μBSML.

μBSML modélise le comportement de BSML en ce qui concerne l'évaluation parallèle mais exclut des aspects de ML utiles en pratique. Nous prévoyons toutefois de les inclure par la suite dans la modélisation en Coq : motifs et filtrage de motifs, références et exceptions, inférence de type [20], modules, *etc.*

La syntaxe abstraite de μBSML est définie par la grammaire de la figure 3 où les opérateurs unaires et binaires sont les suivants :

$$
\begin{array}{rcll}
e & ::= & x & \textit{variable} \\
& | & c & \textit{constante} \\
& | & \textit{unop}\ e & \textit{application d'opérateur unaire} \\
& | & \textit{binop}\ e\ e & \textit{application d'opérateur binaire} \\
& | & \mathsf{fun}\, x \to e & \textit{fonction} \\
& | & e\ e & \textit{application} \\
& | & \mathsf{let}\, x = e\, \mathsf{in}\, e & \textit{définition locale} \\
& | & \mathsf{let\ rec}\, x\ y = e\, \mathsf{in}\, e & \textit{définition récursive} \\
& | & (e, e) & \textit{couple} \\
& | & \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e & \textit{conditionnelle} \\
& | & [e_i]_i^n & \textit{n-tableau} \\
& \multicolumn{3}{l}{\textit{la suite n'est pas accessible à l'utilisateur}} \\
& | & \langle e_i \rangle_i & \textit{vecteur parallèle}
\end{array}
$$

FIGURE 3 – Syntaxe de μBSML

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid e\ \mathrm{E} \mid \mathrm{E}\ e \\
& | & \textit{unop}\ \mathrm{E} \mid \textit{binop}\ e\ \mathrm{E} \mid \textit{binop}\, \mathrm{E}\ e \\
& | & \mathsf{let}\, x = \mathrm{E}\, \mathsf{in}\, e \mid (\mathrm{E}, e) \mid (e, \mathrm{E}) \\
& | & \mathsf{if}\, \mathrm{E}\, \mathsf{then}\, e\, \mathsf{else}\, e \mid [v, \ldots, v, \mathrm{E}, e, \ldots, e]_n
\end{array}
$$

FIGURE 4 – Contextes

$$
\begin{array}{rcll}
\textit{unop} & ::= & \mathsf{mkpar} \mid \mathsf{proj} \mid \mathsf{put} & \textit{primitives parallèles} \\
& | & \mathsf{fst} \mid \mathsf{snd} & \textit{opérateurs de couples} \\
& | & \mathsf{length} & \textit{opérateurs de tableaux} \\
& | & \mathsf{fix} \mid \mathsf{send} & \textit{opérateurs internes} \\
& | & \ldots & \textit{opérations usuelles sur les booléens et les entiers} \\
\textit{binop} & ::= & \mathsf{apply} & \textit{primitive parallèle} \\
& | & \mathsf{nth} & \textit{sur les tableaux} \\
& | & \ldots & \textit{opérations usuelles sur les booléens et les entiers}
\end{array}
$$

Parmi les constantes, nous supposons avoir la valeur (), les valeurs booléennes true et false ainsi que les entiers. Les valeurs de μBSML sont définies par la grammaire suivante :

$$v \quad ::= \quad c \mid \mathsf{fun}\, x \to e \mid (v, v) \mid [v_i]_i^n \mid \langle v_i \rangle_i$$

Comme pour les vecteurs parallèles, on note en abrégé $[\ ]_i^n$ pour désigner le tableau de taille n où l'indice i est lié dans les crochets [] à l'indice de l'élément : $[i]_i^n$ est le tableau $[0, 1, \ldots, (n-1)]$.

Notre sémantique définit une relation de réduction de tête $\rightarrowtail^{\epsilon}$ et un contexte d'évaluation E[·] désignant une expression avec un "trou" unique [·]. Les règles de réduction ne permettant de réduire que des expressions dont les sous-termes sont des valeurs, les contextes permettent de définir, dans le cas d'expressions imbriquées, dans quelle sous-expression il est permis de réduire.

(2) permet la réduction locale sur un processus donné. E_g and E_l correspondent à des contextes quelconques. E_g permet de situer le vecteur parallèle à réduire dans tout le terme et E_l est le contexte dans lequel la composante locale est réduite.

On dispose d'une instance de la règle (2) par processeur, chaque étape de réduction de tête étant choisie de façon non déterministe sur chaque composante du vecteur parallèle jusqu'à ce

$$\frac{e_1 \rightarrowtail^{\epsilon} e_2}{E[e_1] \rightarrowtail E[e_2]} \tag{1}$$

$$\frac{e_1 \rightarrowtail^{\epsilon} e_2}{E_g[\langle e'_0 \dots e'_{i-1}, E_l[e_1], e'_{i+1} \dots e'_{p-1}\rangle] \rightarrowtail E_g[\langle e'_0 \dots e'_{i-1}, E_l[e_2], e'_{i+1} \dots e'_{p-1}\rangle]} \tag{2}$$

FIGURE 5 – BSML classique – Règles de contexte

que le vecteur ne contienne plus que des valeurs, et soit ainsi lui-même considéré comme une valeur.

Cela ne décrit pas le parallélisme de façon réaliste (la réduction est effectuée sur un processeur à la fois). Cependant, le fait qu'on puisse réduire dans un ordre quelconque peut rendre apparent les problèmes d'indéterminisme.

À présent, on donne l'ensemble des règles de réduction de tête du langage qui comportent les δ-règles (règles (8)..(17)) permettant de réduire les opérateurs, et en particulier, les primitives parallèles.

$$(\mathsf{fun}\, x \to e)\ v \rightarrowtail^{\epsilon} \{v/x\}e \tag{3}$$

$$\mathsf{let}\, x = v\, \mathsf{in}\, e \rightarrowtail^{\epsilon} \{v/x\}e \tag{4}$$

$$\mathsf{let\ rec}\, x\ y = e_1\, \mathsf{in}\, e_2 \rightarrowtail^{\epsilon} \mathsf{let}\, x = \mathsf{fix}(\mathsf{fun}\, x \to \mathsf{fun}\, y \to e_1)\, \mathsf{in}\, e_2 \tag{5}$$

$$\mathsf{if}\ \mathbf{true}\ \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \rightarrowtail^{\epsilon} e_1 \tag{6}$$

$$\mathsf{if}\ \mathbf{false}\ \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \rightarrowtail^{\epsilon} e_2 \tag{7}$$

$$\mathsf{mkpar}\, v \rightarrowtail^{\epsilon} \langle v\ i\rangle_i \tag{8}$$

$$\mathsf{apply}\, \langle v_i^1\rangle_i\ \langle v_i^2\rangle_i \rightarrowtail^{\epsilon} \langle v_i^1\ v_i^2\rangle_i \tag{9}$$

$$\mathsf{proj}\, \langle v_i\rangle_i \rightarrowtail^{\epsilon} \mathsf{nth}\, [v_i]_i^p \tag{10}$$

$$\mathsf{put}\, \langle v_i\rangle_i \rightarrowtail^{\epsilon} \mathsf{apply}\ \langle \mathsf{nth}\rangle_i\, (\mathsf{send}\, \left\langle [v_i\ j]_j^p \right\rangle_i) \tag{11}$$

$$\mathsf{send}\, \left\langle [v_j^i]_j^p \right\rangle_i \rightarrowtail^{\epsilon} \left\langle [v_j^i]_i^p \right\rangle_j \tag{12}$$

$$\mathsf{fst}(v_1, v_2) \rightarrowtail^{\epsilon} v_1 \tag{13}$$

$$\mathsf{snd}(v_1, v_2) \rightarrowtail^{\epsilon} v_2 \tag{14}$$

$$\mathsf{length}\, [v_i]_i^n \rightarrowtail^{\epsilon} n \tag{15}$$

$$\mathsf{nth}\, [v_i]_i^n\ k \rightarrowtail^{\epsilon} v_k\ si\ 0 \leq k < n \tag{16}$$

$$\mathsf{fix}(\mathsf{fun}\, x \to e) \rightarrowtail^{\epsilon} \{\mathsf{fix}(\mathsf{fun}\, x \to e)/x\}e \tag{17}$$

5 μBSML$^{\$}$: syntaxe et sémantique révisées

Dans la syntaxe de μBSML$^{\$}$, les opérations **mkpar** et **apply** ne sont plus autorisées. Elles sont remplacées par des notations plus expressive et plus simple $\ll e \gg$ et $\$e\$$. Par conséquent

la syntaxe de μBSML$^{\$}$ est la même que celle de μBSML$^{\$}$ aux différences suivantes près :

$$\begin{array}{rcll} e & ::= & \ldots & \ldots \\ & | & \ll e \gg & \textit{vecteur parallèle} \\ & | & \$e\$ & \textit{projection} \end{array}$$

L'ensemble des opérateurs est lui aussi un peu différent puisqu'on n'a plus besoin d'utiliser apply pour ouvrir un vecteur.

$$\begin{array}{rcll} \textit{unop} & ::= & \mathsf{proj} \mid \mathsf{put} & \textit{primitives parallèles} \\ & | & \mathsf{fst} \mid \mathsf{snd} & \textit{sur les couples} \\ & | & \mathsf{length} & \textit{sur les tableaux} \\ & | & \mathsf{fix} \mid \mathsf{send} & \textit{opérateurs internes} \\ & | & \ldots & \textit{opérations booléennes et arithmétiques usuelles} \\ \textit{binop} & ::= & \mathsf{nth} & \textit{sur les tableaux} \\ & | & \ldots & \textit{opérations booléennes et arithmétiques usuelles} \end{array}$$

Pour accéder au pid local, l'utilisateur dispose d'une valeur prédéfinie this qui est le vecteur d'entiers contenant la valeur i au processeur i. La valeur de cette variable dépend du nombre p de processeurs. this vaut $\langle\ 0\ ,\ldots,\ i\ ,\ldots,\ p-1\ \rangle$.

Finalement, l'ensemble des règles de réduction de tête contient {(3) ... (7)}, (10) et {(12) ... (17)}. Les règles (8) et (9) sont supprimées, et (11) est modifiée comme suit :

$$\mathsf{put}\ \langle v_i \rangle_i \quad \rightarrowtail^{\epsilon} \quad ((\mathsf{fun\ nth} \rightarrow\ (\mathsf{fun}\ v \rightarrow \ll \$\ \mathsf{nth}\ \$\ \$v\$ \gg))\ \langle \mathsf{nth} \rangle_i)\ (\mathsf{send} \left\langle [v_i\ j]_j^p \right\rangle_i) \qquad (18)$$

L'ensemble est complété par la règle de réduction suivante qui développe la notation chevron en répliquant la même expression sur tous les processeurs :

$$\ll e \gg \rightarrowtail^{\epsilon} \langle e, \ldots, e \rangle \qquad (19)$$

L'accès à la valeur locale d'un vecteur ne peut se faire qu'à l'aide de la notation $ $ et dans le contexte d'un vecteur parallèle. La valeur produite dépend du contexte précis dans lequel se fait la réduction, c'est-à-dire la composante du vecteur parallèle :

$$\begin{array}{c} E_g[\langle e_0, \ldots, e_{i-1}, E_l[\$\langle v_0, \ldots, v_i, \ldots, v_{p-1} \rangle \$], e_{i+1}, \ldots, e_{p-1} \rangle] \\ \rightarrowtail \\ E_g[\langle e_0, \ldots, e_{i-1}, E_l[v_i], e_{i+1}, \ldots, e_{p-1} \rangle] \end{array} \qquad (20)$$

La syntaxe des contextes dans BSML révisé ainsi que les règles de contexte sont les mêmes que dans BSML classique (voir respectivement figure 4 et figure 5).

Dans ce qui suit, on illustre par un exemple simple le fait que le formalisme de μBSML$^{\$}$ est plus simple et plus élégant à utiliser du point de vue du programmeur. On définit la fonction broadcast qui effectue la diffusion d'un message à partir d'un processeur root vers tous les autres. La constante no_com signifie qu'aucune transmission n'est effectuée. En BSML classique, on écrit broadcast comme suit :

```
let broadcast root v =
  let msg = put (apply (mkpar (fun i v →
                 if i = root then fun j → v else fun j → no_com)) v )) in
  apply msg (replicate root)
```

La fonction n'est pas aisée à écrire et elle n'est pas non plus facile à comprendre. Si on traduit cette définition en μBSML$^{\$}$, on obtient :

let broadcast root v =
  let msg = **put** $\ll$ **if** $this$=root **then fun** j $\rightarrow$ v **else fun** j$\rightarrow$ no_com $\gg$ **in**
  $\ll$ msg root $\gg$

Si on applique cette fonction à un entier (que l'on suppose être un numéro de processeur valide) root et un vecteur parallèle $\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle$ (que l'on notera par la suite v), on obtient après deux applications de la règle (3) :

let msg = **put** $\ll$ **if** $this$=root **then fun** j $\rightarrow$ v **else fun** j$\rightarrow$ no_com $\gg$ **in**
$\ll$ msg root $\gg$

En utilisant la règle (19), elle se réduit à :

let msg =**put** $\langle$ **if** $this$=root **then fun** j$\rightarrow$ v **else fun** j$\rightarrow$ no_com $,\ldots,$
if $this$=root **then fun** j$\rightarrow$ v **else fun** j$\rightarrow$ no_com $\rangle$ **in**
$\ll$ msg root $\gg$

Après réduction de $this$ par la règle (20) (p fois), on obtient :

let msg =**put** $\langle$ **if** 0=root **then fun** j$\rightarrow$ v **else fun** j$\rightarrow$ no_com $,\ldots,$
if (p$-$1)=root **then fun** j$\rightarrow$ v **else fun** j$\rightarrow$ no_com $\rangle$ **in**
$\ll$ msg root $\gg$

Après réduction de la condition du **if** dans toutes les composantes du vecteur, on peut appliquer la règle (6) au processeur root et la règle (7) dans tous les autres processeurs, et on obtient :

$$\textbf{let msg =put } \langle \textbf{fun j}\rightarrow \text{no\_com}, \ldots, \overbrace{\textbf{fun j}\rightarrow \$v\$}^{\text{root}}, \ldots, \textbf{fun j}\rightarrow \text{no\_com}, \rangle \textbf{ in}$$
$$\ll \$msg\$ \text{ root} \gg$$

En utilisant la règle (18), l'expression se réduit en :

$$\textbf{let msg} = (\ (\textbf{fun f}\rightarrow \textbf{fun x}\rightarrow \ll \$f\$\ \$x\$ \gg)\ \langle \text{nth}\rangle_i\ )$$
$$(\text{send}\ \langle\ [(\textbf{fun j} \rightarrow \text{no\_com})\ \text{j}]^p_j, \ldots, [(\textbf{fun j} \rightarrow \$v\$)\ \text{j}]^p_j, \ldots, [(\textbf{fun j} \rightarrow \text{no\_com})\ \text{j}]^p_j\ \rangle\ )\ \textbf{in}$$
$$\ll \$msg\$ \text{ root} \gg$$

En utilisant $p \times (p-1)$ fois la règle (3) partout excepté au processeur d'indice root où on utilise p fois (3) suivi de (20), on obtient :

$$\textbf{let msg} = (\ (\textbf{fun f}\rightarrow \textbf{fun x}\rightarrow \ll \$f\$\ \$x\$ \gg)\ \langle \text{nth}\rangle_i\ )$$
$$(\text{send}\ \langle\ [\text{no\_com}]^p_j, \ldots, [\text{v}_{\text{root}}]^p_j, \ldots, [\text{no\_com}]^p_j\ \rangle\ )\ \textbf{in}$$
$$\ll \$msg\$ \text{ root} \gg$$

En appliquant la règle (12), l'expression devient :

$$\textbf{let msg} = (\ (\textbf{fun f}\rightarrow \textbf{fun x}\rightarrow \ll \$f\$\ \$x\$ \gg)\ \langle \text{nth}\rangle_i\ )\ \ \langle\ [\text{no\_com}, \ldots, \overbrace{\text{v}_{\text{root}}}^{\text{root}} \ldots, \text{no\_com}]^p\ \rangle_i\ )\ \textbf{in}$$
$$\ll \$msg\$ \text{ root} \gg$$

En utilisant deux fois la règle (3) :

$$\textbf{let msg} = \ll \ \$\langle \mathsf{nth} \rangle_i\$ \ \$\langle \ [\mathsf{no\_com}, \ldots, \mathsf{v_{root}} \ldots, \mathsf{no\_com}]^p \rangle_i\$ \gg \textbf{in}$$
$$\ll \$\mathsf{msg}\$ \ \mathsf{root} \gg$$

Puis avec l'applications de la règle (19) suivie de p applications de la règle (20), on aboutit à l'expression :

$$\textbf{let msg} = \ \langle \mathsf{nth} \ [\mathsf{no\_com}, \ldots, \mathsf{v_{root}} \ldots, \mathsf{no\_com}]^p \rangle_i \ \textbf{in}$$
$$\ll \$\mathsf{msg}\$ \ \mathsf{root} \gg$$

L'application de (4), puis (19), suivi de $2p$ fois (20) conduit à :

$$\langle \mathsf{nth} \ [\mathsf{no\_com}, \ldots, \mathsf{v_{root}} \ldots, \mathsf{no\_com}]^p \ \mathsf{root} \rangle_i$$

Enfin, p applications de la règle (16), donnent la valeur finale :

$$\langle \ \mathsf{v_{root}} \ , \ldots , \ \mathsf{v_{root}} \ , \ldots , \ \mathsf{v_{root}} \ \rangle$$

6 Modélisation et preuves en Coq

Les syntaxes et sémantiques des deux sections précédentes ont été modélisées à l'aide de l'assistant de preuve Coq [1] et les confluences fortes de ces sémantiques ont été établies.

μBSML et μBSML$^\$$ partagent un certain nombre de caractéristiques : les deux syntaxes sont identiques à certaines opérations unaires et binaires prédéfinies près. Un certain nombre de règles de réduction, ainsi que des règles de contexte sont identiques. Il a donc été choisi d'avoir une modélisation modulaire.

6.1 Syntaxe et sémantique génériques

On a ainsi deux bibliothèques Coq GenericSyntax et GenericSemantics qui contiennent chacune un foncteur Make qui prend en argument un module de type OperatorsTypesType qui contient essentiellement le type des opérations unaires et le type des opérations binaires, ainsi qu'un certain nombre de valeurs qu'elles doivent contenir (les opérations ML "séquentielles").

GenericSyntax contient la définition des pré-termes (génériques) qui correspondent à ceux de la figure 3 mais sans les définitions locales. À noter que les paires sont définies comme opération binaire. Les constantes sont celles décrites dans la section 4 : entiers, booléens et valeur de "type unit". En ce qui concerne les variables, nous avons choisi une représentation localement close et avons utilisé la quantification cofinie [4, 13]. Le type preTerm des pré-termes contient ainsi des variables liées (des naturels Coq, indices de de Bruijn) et des variables libres nommées (du type atom fourni par la bibliothèque MetaLib [1]). Une expression est un pré-terme qui est localement clos, c'est-à-dire qui ne contient pas de variable liée qui n'est pas effectivement liée par une abstraction. Dans le cas de BSML, une expression doit de plus avoir la propriété que tous ses vecteurs parallèles énumérés (représentés par des listes de pré-termes) doivent avoir la même taille, p:nat le nombre de processeurs de la machine BSP. Des principes d'induction adaptés, et des fonctions pour manipuler les pré-termes (comme la fonction d'ouverture qui permet de définir la règle de β-contraction), sont définis dans GenericSyntax.

GenericSemantics contient tout d'abord le prédicat inductif qui définit ce qu'est une valeur et quelques résultats et tactiques (par exemple les valeurs sont des expressions). Sont définies ensuite une relation de réduction, ainsi que des opérations sur les relations (de réduction). La réduction générique de tête headReductionML: relation preTerm [2] contient toutes les règles qui

1. `http://www.cis.upenn.edu/~plclub/metalib`
2. où **Definition** relation (A:Type) := A →A →Prop.

ne traitent pas d'opérations parallèles. Toutes les opérations que l'on suppose définies dans les sémantiques des sections précédentes sont définies : opérations arithmétiques, booléennes, comparaisons, ainsi que les règles (3), (6), (7), (13), (14), (15), (16). Dans le cas de (16), on utilise la fonction nth_error de Coq : dans le cas où le résultat est None la règle ne peut pas être appliquée.

Les règles de passage au contexte sont définies par des opérations sur les réductions. La définition inductive

Inductive contextRulesML (R: relation preTerm): relation preTerm

"passe" R aux contextes de la figure 4. On a en effet besoin de passer différentes réductions à ces contextes. De même, on définit

Inductive contextRulePar (pid : nat | pid < p)(R: relation preTerm): relation preTerm

qui modélise la règle de passage au contexte à une composante donnée d'un vecteur parallèle énuméré (règle (2), sans le contexte E_g englobant).

Deux propriétés sont utilisées par la suite :

– la régularité :

```
Definition regularity (R: relation preTerm) :=
  ∀(e1 e2 : preTerm),
    R e1 e2 →
    isExpression p e1 ∧isExpression p e2.
```

– le fait que les valeurs ne peuvent pas être réduites :

```
Definition valuesCannotBeReducedBy (R:relation preTerm):=
  ∀(e : preTerm),
    isValue p e →
    ∀e', not(R e e').
```

Il est montré que headReductionML est régulière et ne peut pas réduire les valeurs, et que ces deux propriétés sont préservées par union et par passage aux contextes contextRulesML et au contexte contextRulePar.

6.2 Syntaxe et sémantique classiques et révisées

La bibliothèque Coq ClassicSemantics applique le foncteur Make de GenericSemantics à un module dans lequel les deux types inductifs des opérateurs unaires et binaires sont définis. Ils contiennent en plus des valeurs imposées par la signature des valeurs spécifiques à μBSML. Pour obtenir la sémantique de la section 4, il suffit de définir la relation headReductionBSML qui modélise les règles (8), (9), (10), (11), (12).

Pour cette dernière règle, on a défini une notion de matrice (une liste de listes qui ont toutes la même taille) et une opération de transposée de matrice : c'est ce que réalise l'opération send si on considère le vecteur (de taille p) de tableaux (de taille p) comme une matrice. La manipulation des valeurs de ce type n'a pas été particulièrement aisée et a nécessité en particulier des preuves de "*proof irrelevance*" (si les paramètres sans leurs preuves sont égaux alors les résultat de l'application de la fonction sans leurs preuves sont égaux) des opérations sur les matrices et les listes de taille donnée.

La réduction de μBSML est alors définie par :

$$\mathsf{contextRulesML}\big(\mathsf{headReduction} \cup \cup_{i=0}^{p-1} \mathsf{contextRulePar}\ i\ (\mathsf{contextRulesML}\ \mathsf{headReduction})\big)$$

où headReduction = headReductionML ∪ headReductionBSML.

La régularité et la non réduction des valeurs est montrée pour ces nouvelles relations.

La modélisation de la sémantique révisée suit le même principe. Toutefois, on ne doit pas passer au contexte des opérations unaires $\ll \cdot \gg$ et $\$\cdot\$$. En réalité la signature OperatorsTypesType contient également deux fonctions qui indiquent quelles opérations permettent que l'on passe au contexte dans leurs arguments ou non. Pour compléter la sémantique générique, deux relations sont définies :

- headReductionBSML qui modélise les règles (10), (18), (12), (19),
- **Inductive** dollarReduction (i:nat |i < p): relation preTerm qui modélise la règle (20).

La réduction de μBSML$^{\$}$ est alors définie par :

$$\mathsf{contextRulesML}\big(\mathsf{headReduction} \quad \cup \\ \cup_{i=0}^{p-1} \mathsf{contextRulePar}\, i\, (\mathsf{contextRulesML}(\mathsf{headReduction} \cup (\mathsf{dollarReduction}\, i)))\big)$$

où headReduction = headReductionML ∪ headReductionBSML.

6.3 Preuves de confluence forte

Pour les deux réductions définies respectivement dans ClassicSemantics et RevisedSemantics, la proposition :

```
Proposition strongConfluence:
  ∀(p:nat),
    isStronglyConfluent (reduction p).
```

est prouvée (respectivement dans Classic et Revised) où :

```
Definition isStronglyConfluent (A:Type)(R:relation A) :=
  ∀(e e1 e2 : A),
    R e e1 →R e e2 →
    exists e', clos_refl R e1 e'∧ clos_refl R e2 e'.
```

Deux lemmes sont préalablement prouvés : la réduction de tête est déterministe et la réduction dans le contexte d'une composante donnée d'un vecteur parallèle est fortement confluente.

Les preuves de ces propositions sont faites par induction sur e en utilisant un principe d'induction adapté au traitement des pré-termes ayant en sous-terme une liste de pré-termes. Les cas où e est une variable (liée ou libre) ou une constante sont éliminés immédiatement.

Les deux hypothèses de réduction sont inversées autant que possible, c'est-à-dire jusqu'à ce que la structure du pré-terme réduit e ne soit plus spécifiée (dans les hypothèses, le but contient un terme qui contient e en sous-terme). On a alors les cas suivants.

- La racine de la réduction de e est d'un côté une réduction de tête et de l'autre une règle de contexte, dans ce cas il y a contradiction car un des sous-termes est une valeur et il est réduit, c'est ici qu'on utilise les résultats sur le fait que les valeurs ne peuvent pas être réduites par nos diverses relations.
- La racine de la réduction de e est des deux côtés une réduction de tête. On utilise le déterminisme de la réduction de tête. On a alors que e1 et e2 sont identiques. Le témoin est le terme qui est déjà dans le but et on conclut par réflexivité de la fermeture réflexive.
- La racine de la réduction est des deux côtés un passage au contexte. On a deux cas.
  - Soit c'est le même sous-terme e qui est réduit. On applique l'hypothèse d'induction (des tactiques ont été élaborées pour construire le témoin en filtrant le but et les hypothèses) puis selon les cas on conclut par réflexivité ou passage au contexte.

– Soit il y a deux sous-termes différents qui sont réduits (cas de l'application ou d'une opération binaire uniquement). On construit le témoin en utilisant ces deux sous-termes. On conclut par réflexivité ou en passant au contexte la réduction appliquée d'un côté de l'autre côté.

Il est à noter que les cas des tableaux et des vecteurs parallèles sont traités à part.

Pour les tableaux, comme la stratégie de réduction est fixée, on montre que forcément la réduction dans le contexte d'un tableau se fait au même indice des deux côtés, sinon il y a une contradiction prouvée à l'aide de la non réduction des valeurs. Il suffit alors de conclure par déterminisme de la réduction de tête.

Pour le contexte d'un vecteur parallèle, soit les réductions sont faites à la même composante du vecteur parallèle et on conclut par la confluence forte de la réduction dans le contexte d'une composante donnée d'un vecteur parallèle, soit elles sont faites à deux processeurs différents. Dans ce cas on décompose la liste représentant le vecteur en cinq morceaux (dont les deux sous-termes réduits) pour pouvoir construire le témoin. Il suffit alors d'appliquer la règle de passage au contexte dans un vecteur parallèle pour conclure.

Les modélisations et les preuves peuvent être téléchargées à l'adresse :

`http://traclifo.univ-orleans.fr/BSML`.

7 Travaux connexes

Pour les langages de programmation data-parallèles, on considère en général un modèle de programmation, la sémantique présentée au programmeur, et un ou des modèles d'exécution qui modélisent ce qui se passe réellement lorsque le programme est exécuté sur une machine parallèle [8]. Les sémantiques présentées ici sont des modèles de programmation de BSML. Il existe également des modèles d'exécution de BSML [24, 18], dont les sémantiques n'ont pas encore été mécanisées. D'autres langages de programmation parallèle de haut-niveau disposent souvent uniquement d'un des deux modèles. On peut citer d'Eden [23], d'OcamlP3L [14] et le calcul de tableaux multi-dimensionnels répartis [16]. [3] présente une sémantique opérationnelle au niveau du modèle de programmation mais décorée par des informations de répartition qui en font un modèle d'exécution. Aucun des travaux précédents ne fait à notre connaissance l'objet de preuves à l'aide d'un assistant de preuve.

Les travaux les plus proches de notre démarche sont ceux autour de ProActive : [10] présente un modèle de parallélisme dont une instanciation pratique est ProActive [5], une bibliothèque de programmation parallèle et distribuée pour Java, au dessus de laquelle a été construite Calcium [12] une bibliothèque de squelettes algorithmiques pour laquelle il existe un système de type et une sémantique opérationnelle [11]. Si la sémantique de Calcium n'est pas mécanisée, la théorie des objets distribués l'est en Isabelle/HOL [21, 22]. Le modèle de parallélisme de ProActive est toutefois très différent de celui de BSML.

8 Conclusions et perspectives

Les architectures parallèles prennent de plus en plus d'importance. Un nombre significatif de travaux se situe dans le contexte des paradigmes de programmation pour les machines parallèles modernes. Dans cet article nous présentons une syntaxe révisée de BSML qui rend plus accessible la programmation parallèle en BSML ainsi que des sémantiques formelles de BSML classique et BSML révisé.

Cette nouvelle syntaxe réduit considérablement la taille du code. Celui-ci est aussi plus simple et plus élégant à lire et sa mise au point est facilitée. L'implantation actuelle de BSML

révisé utilise CamlP4 et la transformation informelle présentée dans la section 3. Par conséquent, la compatibilité du code en découle naturellement et ainsi, il reste possible d'utiliser les programmes BSML prouvés corrects en Coq [17]. Un de nos travaux en cours consiste à prouver la correction de la nouvelle syntaxe/sémantique par rapport à la sémantique classique et établir que la syntaxe révisée est aussi expressive que la syntaxe classique, le tout à l'aide de Coq.

À plus long terme nous envisageons de développer un compilateur certifié de BSML (avec filtrage de motifs parallèle, références, exceptions), tout d'abord vers du code-octet pour une machine parallèle extension de celle d'Objective Caml (dans la lignée de [18]), puis en étendant CompCertML [15].

Références

[1] The Coq Proof Assistant. http ://coq.inria.fr.

[2] The Objective Caml System. http ://www.ocaml.org.

[3] M. Aldinucci and M. Danelutto. Skeleton-based parallel programming : Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4) :179–192, 2007.

[4] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 3–15. ACM, 2008.

[5] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, Deploying, Composing, for the Grid. In J. Cunha and O. F. Rana, editors, *Grid Computing : Software Environments and Tools*. Springer, 2006.

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

[7] R. H. Bisseling. *Parallel Scientific Computation*. Oxford University Press, 2004.

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

[9] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.

[10] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004.

[11] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 45–53. IEEE Computer Society, 2008.

[12] D. Caromel and M. Leyton. Fine tuning algorithmic skeletons. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par Parallel Processing*, volume 4641 of *LNCS 4641*, pages 72–81. Springer, 2007.

[13] A. Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 2010. to appear.

[14] R. D. Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3l 2.0. *Parallel Processing Letters*, 18(1) :149–164, 2008.

[15] Z. Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot, July 2009.

[16] R. Di Cosmo, S. Pelagatti, and Z. Li. A calculus for parallel computations over multi-dimensional dense arrays . *Computer Language Structures and Systems*, 33(3-4) :82–110, 2006.

[17] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. *Technique et Science Informatiques*, 25(10) :1261–1280, 2006.

[18] F. Gava and F. Loulergue. A Parallel Virtual Machine for Bulk Synchronous Parallel ML. In P. M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part I*, number 2657 in LNCS, pages 155–164. Springer Verlag, june 2003.

[19] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5) :665–671, 2005.

[20] L. Gesbert. *Développement systématique et sûreté d'exécution en programmation parallèle structurée*. PhD thesis, University Paris Est, LACL, 2009.

[21] L. Henrio and F. Kammüller. Functional Active Objects : Typing and Formalisation. *ENTCS*, 255 :83–101, 2009.

[22] L. Henrio and M. U. Khan. Asynchronous components with futures : Semantics and proofs in isabelle/hol. *Electr. Notes Theor. Comput. Sci.*, 264(1) :35–53, 2010.

[23] M. Hidalgo-Herrero and Y. Ortega-Mallén. An Operational Semantics for the Parallel Language Eden. *Parallel Processing Letters*, 12(2) :211–228, 2002.

[24] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4) :423–437, 2001.

[25] W. F. McColl. Scalability, portability and predictability : The BSP approach to parallel programming. *Future Generation Computer Systems*, 12 :265–272, 1996.

[26] 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.

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

[28] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8) :103, 1990.