Synthèse de types pour Bulk Synchronous Parallel ML

Frédéric Gava[1] et Frédéric Loulergue[1]

1: Laboratoire d'Algorithmique, Complexité et Logique
Université Parix XII, Val-de-Marne
61, avenue du Général de Gaulle
94010 Créteil cedex – France
{gava,loulergue}@univ-paris12.fr

1. Introduction

Bulk Synchronous Parallel ML ou BSML est une extension de ML pour la programmation fonctionnelle en mode direct d'algorithmes parallèles BSP (*Bulk Synchronous Parallelism*). Le modèle de programmation parallèle BSP a été introduit par Valiant au début des années 90 [21] afin d'offrir à la fois un haut degré d'abstraction comme les modèles PRAM [4] tout en étant portable et en permettant la prévision réaliste de performances sur une grande variété d'architectures. Un algorithme BSP est un algorithme en mode direct [7] lorsque la structure physique de ces processus est explicite. De tels algorithmes offrent des performances prévisibles et portables. Ils peuvent être écrits en BSML à l'aide d'un petit ensemble de primitives issues d'un calcul confluent, le BSλ-calcul [15] :

- un constructeur de vecteurs parallèles,
- une application parallèle asynchrone,
- une opération de communication globale synchrone,
- une conditionnelle globale synchrone.

Notre bibliothèque BSMLlib implante les primitives BSML en Objective Caml [11] avec la bibliothèque MPI (*Message Passing Interface*) [19]. Elle est efficace [14] et ses performances suivent les estimations faites par le modèle de coût (le modèle de coût permet d'estimer les temps d'exécution parallèle) BSP [1].

Cette bibliothèque est utilisée comme base du projet Caraml, qui a pour objectif l'utilisation d'Objective Caml pour la programmation parallèle par grille de calcul, avec des applications par exemple aux bases de données parallèles ou à la simulation moléculaire. Dans un tel contexte, la sécurité est un point très important. Toutefois, la sûreté est un préalable. Une machine abstraite parallèle a été conçue [18, 17] et prouvée correcte par rapport au BSλ_p-calcul [12] en passant par une sémantique intermédiaire [13].

Afin d'avoir à la fois une implantation simple et un modèle de coût qui suit le modèle BSP, l'imbrication de vecteurs parallèles est interdite. BSMLlib étant une bibliothèque, le programmeur est responsable de l'absence d'imbrication. Ceci rompt la sûreté de notre environnement. Cet article présente un système de types qui empêche ce genre d'imbrications (section 5) et un algorithme de synthèse de types (section 6). Nous présentons tout d'abord le modèle BSP (section 2), donnons une présentation informelle de BSML (section 3) et expliquons de façon plus détaillée les raisons qui nous poussent à refuser l'imbrication de vecteurs parallèles (section 4).

2. Le modèle BSP

Le modèle de programmation parallèle BSP (*Bulk Synchronous Parallelism*) [16] décrit une architecture parallèle (abstraite), un modèle d'exécution et un modèle de coût.

2.1. L'architecture parallèle BSP

Un ordinateur parallèle BSP possède trois ensembles de composants :

- un ensemble homogène de paires processeur-mémoire,
- un réseau de communication permettant l'échange de messages entre chaque couple de processeurs,
- une unité de synchronisation globale qui exécute des demandes collectives de barrières de synchronisation.

De nombreuses architectures réelles peuvent être vues comme des ordinateurs parallèles BSP. Par exemple, les machines à mémoire partagée peuvent être utilisées de telle sorte que chaque processeur n'accède qu'à une partie (qui sera alors "privée") de la mémoire partagée et les communications peuvent être faites en utilisant des zones de la mémoire partagée réservées à cet usage. De plus, l'unité de synchronisation est rarement physique mais plutôt logicielle ([10] présente plusieurs algorithmes à cet effet).

Les performances d'un ordinateur BSP sont caractérisées par trois paramètres (exprimés en multiples de la vitesse des processeurs, dans le cas contraire un quatrième paramètre, la vitesse des processeurs est donnée) :

- le nombre de paires processeur-mémoire p,
- le temps l nécessaire à la réalisation d'une barrière de synchronisation,
- le temps g pour un échange collectif de messages, appelé 1-relation entre les différentes paires processeur-mémoire dans laquelle chaque processeur envoie et/ou reçoit au plus un mot ; le réseau peut réaliser un échange, appelé h-relation (chaque processeur envoie et/ou reçoit au plus h-mots) en temps $h \times g$.

Ces paramètres peuvent être facilement obtenus en pratique en utilisant des tests [9].

2.2. Le modèle d'exécution

L'exécution d'un programme BSP est une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases sucessives et logiquement disjointes (Fig. 1) :

- chaque processeur utilise les données qu'il détient localement pour faire des calculs de façon séquentielle et pour demander des tranferts depuis ou vers d'autres processeurs,
- le réseau réalise les échanges de données demandés à la phase précédente,
- une barrière de synchronisation globale termine la super-étape. À l'issue de cette barrière de synchronisation globale, les données échangées sont effectivement disponibles pour la nouvelle super-étape qui commence alors.

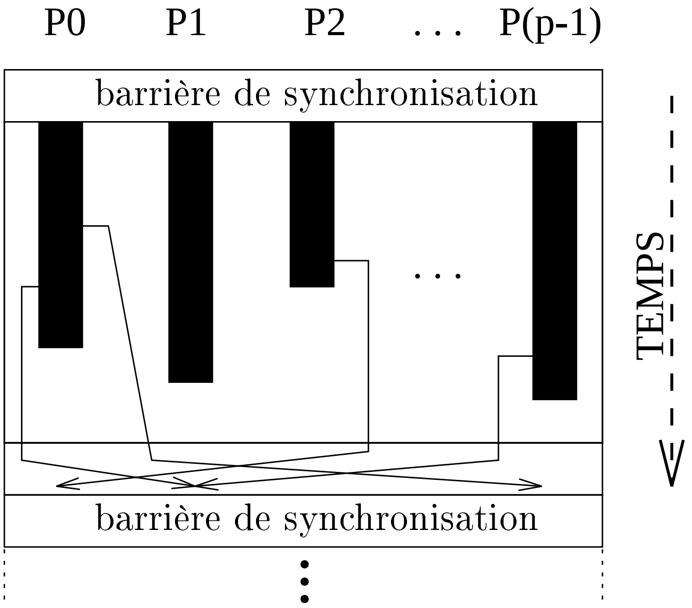

FIG. 1: *Une super-étape BSP*

2.3. Le modèle de coût

Le temps nécessaire à l'exécution d'une super-étape s est la somme :

- du maximum des temps de calculs locaux,
- du temps de la réalisation des échanges entre processeurs,
- du temps de la réalisation d'une barrière de synchronisation globale.

On l'exprime par la formule suivante :

$$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processeur} h_i^{(s)} \times g + l$$

où $w_i^{(s)}$ est le temps de calcul local sur le processeur i pendant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots envoyés (resp. reçus) par le processeur i durant la super-étape s.

Le temps d'exécution $\sum_s \text{Time}(s)$ d'un programme BSP composé de S super-étapes est donc une somme de trois termes :

$$W + H \times g + S \times l$$

où $\left\{ \begin{array}{lcl} W & = & \sum_s \max_i w_i^{(s)} \\ H & = & \sum_s \max_i h_i^{(s)}. \end{array} \right.$

En général W, H et S sont fonctions de p et de la taille des données n, ou de paramètres plus complexes tels que le déséquilibre des données (*data skew*). Afin de minimiser le temps d'exécution, un algorithme BSP doit à la fois minimiser le nombre de super-étapes, le volume total H et le déséquilibre des communications, le volume total W et le déséquilibre des calculs locaux.

3. La bibliothèque BSMLlib

Il n'y a pas pour l'instant d'implantation complète du langage Bulk Synchronous Parallel ML mais une implantation partielle en tant que bibliothèque pour Objective Caml, appelée BSMLLIB.

En premier lieu cette bibliothèque donne accès aux paramètres BSP de l'architecture sur laquelle sont évalués les programmes BSML. En particulier, elle offre la fonction `bsp_p:unit->int`. La valeur de `bsp_p()` est p, le nombre statique de processeurs de la machine parallèle. Cette valeur est constante durant l'exécution.

Les valeurs parallèles de largeur p contenant en chaque processeur une valeur de type `'a` sont représentées par le type abstrait `'a par`. L'imbrication de vecteurs parallèles est interdite. Jusqu'à présent le programmeur était responsable de l'absence d'imbrication. Le système de types et l'algorithme de synthèse de types présentés dans cet article remédient à ce défaut. Ceci constitue une amélioration par rapport à Caml Flight [8, 3] dans lequel l'imbrication de la structure parallèle globale de contrôle `sync` était interdite *dynamiquement*.

Les vecteurs parallèles sont créés par :

```
mkpar: (int ->'a) ->'a par
```

`(mkpar f)` s'évalue en un vecteur parallèle qui possède au processeur `i` la valeur de `(f i)`, pour tout i compris entre 0 et $(p-1)$. Nous écrivons souvent `fun pid->e` pour `f` afin de montrer que `e` peut être différente sur chaque processeur. Cette expression `e` est dite *locale*. L'expression `(mkpar f)` est un objet parallèle et est dite *globale*.

Un algorithme BSP est exprimé comme une combinaison de calculs locaux asynchrones (première phase d'une super-étape) de communications globales (seconde phase d'une super-étape) et d'une synchronisation (troisième phase d'une super-étape). Les calculs asynchrones sont programmés avec `mkpar` et avec :

```
apply: ('a ->'b) par ->'a par ->'b par
```

`apply (mkpar f) (mkpar e)` s'évalue en un vecteur parallèle qui contient `(f i) (e i)` au processeur `i`. Ni l'implantation de BSMLLIB, ni sa sémantique [13] ne préconisent de synchronisation entre deux appels successifs à `apply`.

Nous ignorons la distinction entre la phase de demande de communications et sa réalisation à la barrière de synchronisation. Les phases de communications et de synchronisation sont exprimées à l'aide de :

```
put:(int->'a option) par ->(int->'a option) par
```

où `'a option` est définie par `type 'a option = None | Some of 'a`.

Considérons l'expression :

$$\texttt{put(mkpar(fun i->fs}_i\texttt{))} \tag{1}$$

Pour envoyer une valeur `v` d'un processeur `j` vers un processeur `i`, la fonction $\texttt{fs}_j$ du processeur `j` doit être telle que $(\texttt{fs}_j\ \texttt{i})$ s'évalue en `Some v`. Pour ne pas envoyer de message de `j` à `i`, $(\texttt{fs}_j\ \texttt{i})$ doit s'évaluer en `None`.

L'expression (1) s'évalue en un vecteur parallèle contenant en chaque processeur une fonction $\texttt{fd}_i$ de messages transmis. Au processeur `i`, $(\texttt{fd}_i\ \texttt{j})$ s'évalue en `None` si le processeur `j` n'a pas envoyé de message à `i` ou s'évalue en `Some v` si le processeur `j` a envoyé la valeur `v` au processeur `i`.

Le langage complet contiendra également une conditionnelle globale synchrone :

```
ifat: (bool par) * int * 'a * 'a ->'a
```

telle que `ifat (v,i,v1,v2)` s'évaluera en `v1` ou `v2` selon que la valeur de `v` au processeur `i` est `true` ou `false`. Objective Caml étant un langage strict avec appel par valeur, cette conditionnelle ne peut être définie comme une fonction. C'est la raison pour laquelle BSMLLIB contient la fonction `at:bool par ->int ->bool` qui doit être utilisée uniquement dans la construction suivante : `if (at vec pid) then... else...` où `(vec:bool par)` et `(pid:int)` et dont la sémantique est celle de `ifat`. Cette conditionnelle globale permet d'exprimer des phases de communications et synchronisation. Sans elle, il est impossible d'écrire des algorithmes ayant la forme suivante :

```
Repeat
  Parallel Iteration
Until Max of local errors <epsilon
```

4. Motivations

Nous présentons dans cette section les raisons pour lesquelles nous refusons l'imbrication de vecteurs parallèles dans notre langage.

Considérons le programme BSML suivant :

```
bcast: int->'a par->'a par
let bcast n vec =
 let tosend=mkpar (fun pid->fun v->fun dst->
                      if pid=n then Some v else None) in
   let recv=put(apply tosend vec) in
     apply (replicate noSome) (apply recv (replicate n))
```

Ce programme utilise les fonctions suivantes :

```
replicate: 'a -> 'a par
let replicate x = mkpar(fun pid->x)
noSome: 'a option -> 'a
let noSome (Some x) = x
```

`(bcast 2 vec)` diffuse la valeur du vecteur parallèle `vec` qui se trouve au processeur 2 à tous les autres processeurs. Le coût BSP d'un appel à cette fonction est :

$$p \;+\; (p-1)\times s\times g \;+\; l \tag{2}$$

où s est la taille de la valeur de `vec` se trouvant sur le processeur 2.

Considérons maintenant l'expression suivante :

```
let example1 = mkpar(fun pid->bcast pid vec)
```

Son type est τ `par par` où τ est le type des composants du vecteur parallèle `vec`. La difficulté est de donner un sens en terme de parallélisme à cette expression. Dans la section 3, nous avons indiqué que `mkpar f` s'évalue en un vecteur parallèle qui contient la valeur de `f i` au processeur `i`. Dans cet exemple, le processeur `0` devrait donc contenir la valeur de l'expression `(bcast 0 vec)` qui est une valeur parallèle. Notre langage étant basé sur un calcul confluent [15], les expressions peuvent aussi bien être évaluées en parallèle que de façon séquentielle. Il est ainsi tout à fait possible que l'expression `bcast 0 vec` soit évaluée séquentiellement par le processeur `0`. Toutefois dans ce cas, le temps d'exécution ne sera pas du tout conforme à la formule (2) de coût BSP. Le coût d'évaluation d'une expression sera alors dépendant du contexte, rendant le modèle de coût non compositionnel, ce que nous voulons éviter.

Une autre possibilité serait d'évaluer l'expression `bcast 0 vec` possédée par le processeur `0` en parallèle. Il faudrait pour cela une diffusion préalable de cette expression sur tous les autres processeurs. Une fois cette diffusion réalisée l'évaluation de l'expression suivrait la formule (2) de coût BSP. Toutefois cette phase préalable de diffusion nécessite évidemment des communications et synchronisations. Le coût additionnel rend à nouveau le modèle de coût non compositionnel puisque l'expression `bcast 0 vec` évaluée en dehors d'un `mkpar` ne nécessite pas cette phase préalable. De plus ce choix impliquerait l'usage d'un ordonnanceur et rendrait les formules de coûts particulièrement malaisées à écrire.

Afin d'éviter ces problèmes, l'imbrication de vecteurs parallèles n'est pas autorisée. Le typage de programmes ML est bien connu mais n'est pas adapté à notre langage. En effet, l'imbrication de vecteurs parallèles n'est pas toujours aussi évidente (c'est-à-dire visible dans le type) que dans l'exemple précédent.

Considérons le programme suivant :

```
let example2=
  mkpar(fun pid->
         let this=mkpar(fun pid->pid)
        in pid)
```

Son type est `int par`, donc a priori sans imbrication de vecteurs parallèles, mais son évaluation conduit à celle du vecteur parallèle `this` à l'intérieur du vecteur parallèle englobant. Ainsi nous avons une imbrication de vecteurs parallèles sans que cela soit visible dans le type de l'expression.

Un autre problème arrive avec les valeurs polymorphes. Le cas le plus simple illustrant cette difficulté est le cas d'une projection :

```
let fst = fun (a,b) -> a
```

Son type est bien sûr `'a * 'b ->'a`. Le problème est que certaines de ses instanciations ne sont pas des expressions correctes de BSML. Donnons quatre cas d'application de `fst` à des valeurs de genres différents (habituelles ou parallèles) :

1. deux valeurs habituelles : `fst(1,2)`

2. deux valeurs parallèles : `fst ( mkpar(fun i ->i), mkpar(fun i ->i) )`

3. parallèle et habituelle : `fst (mkpar(fun i ->i), 1)`

4. habituelle et parallèle : `fst (1, mkpar(fun i ->i))`

Le cas posant problème est le quatrième cas. Son type donné par le système de types ML est `int`. Toutefois son évaluation dans le contexte d'un `mkpar` conduirait comme l'exemple `example1` à une imbrication de vecteurs parallèles.

L'objectif de notre système de types est donc de rejeter ces expressions.

5. Un système de types polymorphes pour BSML

5.1. L'algèbre des types

Les types simples sont définis par la grammaire suivante :

$$\begin{array}{rcll} \tau & ::= & \kappa & \text{type de base (bool, int, unit etc.)} \\ & | & \alpha & \text{variable de type} \\ & | & \tau_1 \to \tau_2 & \text{fonction de } \tau_1 \text{ vers } \tau_2 \\ & | & \tau_1 * \tau_2 & \text{couple} \\ & | & (\tau\ par) & \text{vecteur parallèle} \end{array}$$

Trois sous-ensembles de types simples nous intéressent plus particulièrement :

– l'ensemble L des types "locaux" qui représentent les types Caml. Les types locaux (notés $\dot{\tau}$) sont donnés par la grammaire suivante :

$$\dot{\tau} ::= \kappa \mid \dot{\tau}_1 \to \dot{\tau}_2 \mid \breve{\tau} \to \dot{\tau} \mid \dot{\tau}_1 * \dot{\tau}_2$$

– l'ensemble V des types "avec variables" (dont on ne sait pas si l'instatiation est locale ou non). Les types avec variables (notés $\breve{\tau}$) sont donnés par la grammaire suivante :

$$\breve{\tau} ::= \alpha \mid \dot{\tau}_1 \to \breve{\tau}_2 \mid \breve{\tau}_1 \to \breve{\tau}_2 \mid \breve{\tau}_1 * \breve{\tau}_2 \mid \breve{\tau}_1 * \dot{\tau}_2 \mid \dot{\tau}_1 * \breve{\tau}_2$$

– l'ensemble G des types "globaux" qui représentent les objets parallèles. Les types globaux (notés $\bar{\tau}$) sont donnés par la grammaire suivante :

$$\begin{array}{rcl} \bar{\tau} & ::= & (\breve{\tau}\ par) \mid (\dot{\tau}\ par) \mid \breve{\tau}_1 \to \bar{\tau}_2 \mid \dot{\tau}_1 \to \bar{\tau}_2 \mid \bar{\tau}_1 \to \breve{\tau}_2 \mid \bar{\tau}_1 \to \bar{\tau}_2 \\ & & \mid \bar{\tau}_1 * \bar{\tau}_2 \mid \breve{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \breve{\tau}_2 \mid \dot{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \dot{\tau}_2 \end{array}$$

Remarque : Nous avons $L \cap G = \emptyset$, $V \cap G = \emptyset$ et $V \cap L = \emptyset$.

Remarque : L'extention native des N-uplets n'a pas été ajouté car même si elle n'est pas difficile en soit, elle alourdit considérablement les notations.

Le problème évoqué à la section 4 est que toute instanciation d'un type avec variable n'appartient pas nécessairement à $L \cup V \cup G$. Prenons par exemple le type simple $(\alpha\ par) \to int$ ou le type simple $(\alpha\ par)$ et l'instanciation $\alpha = (int\ par)$: nous avons alors une imbrication de vecteurs parallèles.

Afin d'éviter ce genre de problème, nous utiliserons des contraintes indiquant quelles variables peuvent être instanciées par des types dans L ou non.

Nous introduisons d'abord la notion de schéma de type avec contraintes :

$$\sigma ::= \forall \alpha_1 ... \alpha_n . [\tau / C]$$

où τ est un type simple et C est une contrainte du calcul propositionnel classique donnée par la grammaire suivante :

$$\begin{array}{rcl} C & ::= & \textbf{False} \\ & | & \textbf{True} \\ & | & \mathcal{L}(\alpha) \\ & | & C_1 \wedge C_2 \\ & | & C_1 \Rightarrow C_2 \end{array}$$

Lorsque la quantification est vide, nous notons simplement $[\tau/C]$ et nous n'écrivons pas les contraintes lorsqu'elles sont **True**. Nous travaillons modulo les équations suivantes : **True** $\wedge\ C\ =\ C$, $C\ \wedge\ C\ =\ C$ et la commutativité de l'opérateur $\wedge$.

Pour un type simple τ, $\mathcal{L}(\tau)$ indique que ce type est dans L. Pour décomposer cette contrainte en contraintes sur les variables de type, on utilise les égalités qui suivent :

$$\begin{array}{rcll}
\mathcal{L}(\alpha) & = & \textbf{True} \quad \text{si } \alpha \in \kappa & (Kappa) \\
\mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2) & (Arrow) \\
\mathcal{L}(\tau_1 * \tau_2) & = & \mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2) & (Couple) \\
\mathcal{L}(\tau\ par) & = & \textbf{False} & (Par)
\end{array}$$

Dans le système de types et pour la substitution d'un schéma de type, nous utiliserons des contraintes appelées contraintes de base construites à partir du type simple seul. On note C_τ les contraintes de base associées au type simple τ, définies par :

$$\begin{array}{lcll}
C_\tau & = & \textbf{True} \quad \text{si } \tau \text{ atomic} & (CAtom) \\
C_{(\tau_1 \to \tau_2)} & = & C_{\tau_1} \wedge C_{\tau_2} \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1) & (CArrow) \\
C_{(\tau_1 * \tau_2)} & = & C_{\tau_1} \wedge C_{\tau_2} & (CCouple) \\
C_{(\tau\ par)} & = & \mathcal{L}(\tau) & (CPar)
\end{array}$$

Les contraintes manipulées ajoutent à ces contraintes de base des contraintes sur les sous-expressions, qui sont indispensables au traitement des cas similaires à l'exemple `example2` de la section 4.

L'ensemble des variables libres d'un schéma de type est défini par :

$$\mathcal{F}(\forall \alpha_1 \ldots \alpha_n.[\tau/\mathcal{C}]) \quad = \quad (\mathcal{F}(\tau) \cup \mathcal{F}(C)) \setminus \{\alpha_1, \ldots, \alpha_n\}$$

où les variables libres du type et des contraintes sont définies de façon classique par induction structurelle. Nous notons *Dom* le domaine d'une substitution sur un type simple.

Définition 1 *Une substitution sur un schéma de type est définie par :*

$$\begin{array}{c}
\varphi(\forall \alpha_1 \ldots \alpha_n.[\tau/C]) \\
= \\
\forall \alpha_1 \ldots \alpha_n.[\varphi(\tau)/\varphi(C) \displaystyle\bigwedge_{\beta_i \in Dom(\varphi) \cap \mathcal{F}([\tau/C])} C_{\varphi(\beta_i)}]
\end{array}$$

si $\alpha_1 \ldots \alpha_n$ sont hors de portée de φ.

Une variable de type α est hors de portée d'une substitution φ si $\varphi(\alpha) = \alpha$ (φ ne modifie pas α ou α n'est pas dans le domaine de φ) et si α n'est pas libre dans $[\tau/C]$, alors α ne doit pas être libre dans $\varphi([\tau/C])$ (φ n'introduit pas α dans le résultat). La condition "$\alpha_1 \ldots \alpha_n$ sont hors de portée de φ" peut toujours être obtenue en renommant $\alpha_1 \ldots \alpha_n$ par des variables fraîches dont nous supposons avoir un ensemble infini.

5.2. Instanciation et généralisation

Un schéma de type peut être vu comme l'ensemble des types obtenus par instanciation des variables quantifiées.

Définition 2 *$[\tau/C]$ est une instanciation d'un schéma de type avec contraintes $\forall \alpha_1 ... \alpha_n.[\tau'/C']$, noté $[\tau/C] \leq \forall \alpha_1 ... \alpha_n.[\tau'/C']$, si et seulement si il existe une substitution φ de domaine $\alpha_1, \ldots, \alpha_n$ telle que $[\tau/C] = \varphi([\tau'/C'])$.*

$TC(i)$	$=$	$\mathbf{int} \quad i = 0, 1, \ldots$
$TC(b)$	$=$	$\mathbf{bool} \quad b = \mathbf{true}, \mathbf{false}$
$TC(())$	$=$	$\mathbf{unit}$
$TC(+)$	$=$	$(\mathbf{int} * \mathbf{int}) \to \mathbf{int}$
$TC(\mathbf{fix})$	$=$	$\forall \alpha.(\alpha \to \alpha) \to \alpha$
$TC(\mathbf{fst})$	$=$	$\forall \alpha\beta.[(\alpha * \beta) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]$
$TC(\mathbf{snd})$	$=$	$\forall \alpha\beta.[(\alpha * \beta) \to \beta / \mathcal{L}(\beta) \Rightarrow \mathcal{L}(\alpha)]$
$TC(\mathbf{ifthenelse})$	$=$	$\forall \alpha.(\mathbf{bool} * \alpha * \alpha) \to \alpha$
$TC(\mathbf{mkpar})$	$=$	$\forall \alpha.[(\mathbf{int} \to \alpha) \to (\alpha\ par) / \mathcal{L}(\alpha)]$
$TC(\mathbf{apply})$	$=$	$\forall \alpha\beta.[((\alpha \to \beta)\ par * (\alpha\ par)) \to (\beta\ par) / \mathcal{L}(\alpha) \wedge \mathcal{L}(\beta)]$
$TC(\mathbf{ifat})$	$=$	$\forall \alpha.[(\mathbf{int} * (\mathbf{bool}\ par) * \alpha * \alpha) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathbf{False}]$
$TC(\mathbf{put})$	$=$	$\forall \alpha.[(\mathbf{int} \to \alpha)\ par \to (\mathbf{int} \to \alpha)\ par / \mathcal{L}(\alpha)]$
$TC(\mathbf{nc})$	$=$	$\forall \alpha.\mathbf{unit} \to \alpha$
$TC(\mathbf{isnc})$	$=$	$\forall \alpha.[\alpha \to \mathbf{bool} / \mathcal{L}(\alpha)]$

FIG. 2: *Définition de TC*

Nous notons E un environnement qui associe des schémas de types aux variables libres d'une expression. Son domaine est noté $Dom(E) = \{x_1, \ldots, x_n\}$. Nous supposons que tous les identificateurs sont distincts. L'environnement vide est noté $\emptyset$ et $E(x)$ désigne le schéma de type associé à x par E. Une substitution φ sur E est une substitution point à point sur le domaine de E. L'ensemble des variables libres est naturellement défini par les variables libres des schémas de types associées au domaine de E.

Nous notons $E + \{x : \sigma\}$ l'extension de E par l'association de x à σ. Si, avant cette opération, nous avions $x \in Dom(E)$, nous remplaçons $E(x)$ par le nouveau schéma de type σ associé à x.

Définition 3 *Étant donnés un environnement E, un schéma de type τ sans quantification universelle, nous définissons l'opérateur Gen permettant d'introduire la quantification universelle par :*

$$Gen([\tau/C], E) = \forall \alpha_1 ... \alpha_n.[\tau/C]$$

où $\{\alpha_1, ..., \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E)$

Gen permet l'introduction du polymorphisme.

5.3. Règles de typage

Nous notons TC (figure 2) la fonction qui associe un schéma de type aux constantes et aux primitives. Les primitives **nc** et **isnc** sont utilisées pour simuler le type somme `option` d'Objective Caml; **nc** joue le rôle de fonction constante de non-communication et **isnc** de test de cette même constante.

Nous formulons l'inférence de type par un ensemble de règles qui associent un type à une expression du langage. Le contexte dans lequel une expression est associée à un type est representé par un environnement qui associe des identificateurs à des schémas de types. Les conclusions sont de la forme $E \vdash e : [\tau/C]$. La sémantique statique manipule les schémas de type en utilisant les mécanismes de généralisation et d'instanciation donnés dans les sections précédentes. Le système est donné à la figure 3.

Dans toutes les règles de la figure 3, si une contrainte C est telle que $Solve(C) = \mathbf{False}$ alors la règle ne peut être appliquée et l'expression n'est pas bien typée. *Solve* est définie en utilisant les règles classiques de réduction booléenne du calcul propositionnel et les égalités transformant la localité

$$\frac{[\tau/C] \leq E(x)}{E \vdash x : [\tau/C]}(Var)$$

$$\frac{[\tau/C] \leq TC(c)}{E \vdash c : [\tau/C]}(Const)$$

$$\frac{[\tau/C] \leq TC(op)}{E \vdash op : [\tau/C]}(Op)$$

$$\frac{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e : [\tau_2/C_2]}{E \vdash (\mathbf{fun}\ x \to e) : [\tau_1 \to \tau_2/C_{(\tau_1 \to \tau_2)} \wedge C_2]}(Fun)$$

$$\frac{E \vdash e_1 : [\tau' \to \tau/C_1] \qquad E \vdash e_2 : [\tau'/C_2]}{E \vdash (e_1\ e_2) : [\tau/C_1 \wedge C_2]}(App)$$

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E + \{x : Gen([\tau_1/C_1], E)\} \vdash e_2 : [\tau_2/C_2]}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : [\tau_2/C_1 \wedge C_2 \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)]}(Let)$$

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E \vdash e_2 : [\tau_2/C_2]}{E \vdash (e_1, e_2) : [\tau_1 * \tau_2/C_1 \wedge C_2]}(Pair)$$

FIG. 3: *Règles de typage*

d'un type en contraintes. Nos contraintes étant un sous ensemble du calcul propositionnel, *Solve* est décidable.

De façon classique, (Op), $(Const)$ et (Var) utilisent l'instanciation d'un schéma de type.

La règle (Fun) ajoute un nouveau schéma de type à l'environnement dont la contrainte est une contrainte de base issue du type simple.

Les règles (App), $(Couple)$ et (Let) utilisent la conjonction de contraintes. De plus dans la règle (Let), nous introduisons la contrainte $\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)$ car une expression telle que $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ peut être vue comme $(\mathbf{fun}\ x \to e_2)\ e_1$ et nous devons interdire les expressions de valeurs globales vers des valeurs locales qui conduisent aux problèmes rencontrés dans l'exemple `example2` de la section 4.

Les contraintes de base sont importantes dans notre système mais ne sont pas suffisantes. L'utilisation de contraintes associées aux sous-expressions est indispensable. Considérons par exemple l'expression :

```
fun x -> ifat ((mkpar (fun i -> true)),(0,(x,x)))
```

Le type donné par la synthèse de type Objective Caml est $\alpha \to \alpha$ et les contraintes de base sont $(\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\alpha))$, et sont toujours résolues en **True**. Il est toutefois facile de constater que la variable x (de type α) ne doit pas être instanciée par un type local. Notre système de type, en utilisant les contraintes associées aux sous-expressions (ici **ifat**) donne le schéma de type : $[\alpha \to \alpha/\mathcal{L}(\alpha) \Rightarrow \mathbf{False}]$, qui explicite l'interdiction de l'instanciation de α par un type local (i.e. α ne peut être local et donc son instantiation est dans G).

Remarque : les systèmes de types avec contraintes ont été souvent étudiés dans un cadre général nommé HM($\mathcal{X}$) [23]. Mais notre système n'est pas un sous-cas de ce système car, par exemple, l'abstraction de fonctions engendre des contraintes supplémentaires. De plus ce système de types a été prouvé correct vis-à-vis du λ-calcul et pas du BSλ-calcul

6. Un algorithme de synthèse de types

L'existence d'un algorithme d'inférence de type est un trait nécessaire à un compilateur BSML. Pour les expressions ML, cet algorithme est bien connu [2]. Nous adaptons cet algorithme à notre système de types.

6.1. Présentation de l'algorithme W

6.1.1. Préliminaires

Instance triviale Pour commencer, nous définissons la notion d'instance triviale $Inst(\sigma, V)$ d'un schéma de type :

$$Inst(\forall \alpha_1, ..., \alpha_n.[\tau/\mathcal{C}], V) = \Big(\ [\tau[\alpha_i \leftarrow \beta_i] \ / \ \mathcal{C}[\alpha_i \leftarrow \beta_i]] \ \ , \ \ V \setminus \{\beta_1, ..., \beta_n\} \ \Big)$$

où $\beta_1, ..., \beta_n$ sont n variables distinctes de V, ensemble infini de variables fraîches.

Unification de types Une substitution φ est un unificateur de deux types τ_1, τ_2, si nous avons $\varphi(\tau_1) = \varphi(\tau_2)$. Deux types peuvent être unifiés s'il existe un unificateur pour ces deux types. Un unificateur φ de τ_1, τ_2 est plus général, si tout autre unificateur ψ de τ_1, τ_2, peut être décomposé en $\theta \circ \varphi$ pour une substitution θ. L'unificateur principal, s'il existe, représente les modifications minimales qui permettent d'unifier ces deux types. Tout autre unificateur doit faire au moins ces modifications. L'algorithme W nécessite une unification de premier ordre.

Proposition 1 *Si deux types τ_1 et τ_2 peuvent être unifiés, alors il existe un unique unificateur principal, noté $mgu(\tau_1, \tau_2)$.*

L'algorithme de Robinson adapté à notre algèbre de types prend un ensemble d'équations entre les types et donne l'unificateur principal :

$$\begin{array}{rcl}
Rob(\emptyset) & = & id \\
Rob(\{\alpha \stackrel{?}{=} \alpha\} \cup R) & = & Rob(R) \\
Rob(\{\alpha \stackrel{?}{=} \tau\} \cup R) & = & Rob(R[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \\
Rob(\{\tau \stackrel{?}{=} \alpha\} \cup R) & = & Rob(R[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \\
Rob(\{(\tau_1 \to \tau_2) \stackrel{?}{=} (\tau_1' \to \tau_2')\} \cup R) & = & Rob(\{\tau_1 \stackrel{?}{=} \tau_1'\} \cup \{\tau_2 \stackrel{?}{=} \tau_2'\} \cup R) \\
Rob(\{(\tau_1 * \tau_2) \stackrel{?}{=} (\tau_1' * \tau_2')\} \cup R) & = & Rob(\{\tau_1 \stackrel{?}{=} \tau_1'\} \cup \{\tau_2 \stackrel{?}{=} \tau_2'\} \cup R) \\
Rob(\{(\tau_1 \ par) \stackrel{?}{=} (\tau_1' \ par)\} \cup R) & = & Rob(\{\tau_1 \stackrel{?}{=} \tau_1'\} \cup R)
\end{array}$$

Dans les autres cas, Rob échoue et il n'y a pas d'unificateur. Nous notons $mgu(\tau_1, \tau_2) = Rob(\{\tau_1 \stackrel{?}{=} \tau_2\})$.

6.1.2. L'algorithme W

L'algorithme W prend en entrée une expression syntaxique valide de notre langage et donne le type de cette expression (avec contraintes) et une substitution pour l'environnement. Ce type sera le type le plus général de l'expression.

- **Entrée**: un environnement E, une expression e, un ensemble V de variables fraîches.

- **Sortie**: $([\tau/C], \varphi, V')$ où τ est le type inféré, φ est la substitution à faire sur les variables libres de E, V' est V sans les variables que W utilise et $Solve(C) \neq$ **False**.

Variable : si e est une variable x telle que $x \in Dom(E)$ alors $([\tau/C], V') = Inst(E(x), V)$ et $\varphi = id$.

Constante/Opérateur : si e est une constante c ou un opérateur op alors
$([\tau/C], V') = Inst(TC(e), V)$ et $\varphi = id$.

Abstraction : si e est une abstraction **fun** $x \to e_1$ alors étant donnés $\alpha \in V$ et
$([\tau_1/C_1], \varphi_1, V_1) = W(E + \{x : \alpha\}, e_1, V \setminus \{\alpha\})$ alors
soit $C = C_{\varphi_1(\alpha) \to \tau_1} \wedge C_1$, si $Solve(C) \neq$ **False** alors
$\tau = \varphi_1(\alpha) \to \tau_1$, et $\varphi = \varphi_1$ et $V' = V_1$.

Application : si e est une application $(e_1\ e_2)$ alors étant donnés
$([\tau_1/C_1], \varphi_1, V_1) = W(E, e_1, V)$ et
$([\tau_2/C_2], \varphi_2, V_2) = W(\varphi_1(E), e_2, V1)$ et
étant donnés $\alpha \in V_2$ et $\mu = mgu\{\varphi_2(\tau_1) = \tau_2 \to \alpha\}$ et
$[\tau_1'/C_1'] = \mu \circ \varphi_2([\tau_1/C_1])$ et $[\tau_2'/C_2'] = \mu([\tau_2/C_2])$ alors
soit $C = C_1' \wedge C_2'$ et si $Solve(C) \neq$ **False** alors
$\tau = \mu(\alpha)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V' = V_2 \setminus \{\alpha\}$.

Couple : si e est un couple (e_1, e_2) alors étant donnés
$([\tau_1/C_1], \varphi_1, V_1) = W(E, e_1, V)$ et
$([\tau_2/C_2], \varphi_2, V_2) = W(\varphi_1(E), e_2, V_1)$ et
$[\tau_1'/C_1'] = \varphi_2([\tau_1/C_1])$ alors
soit $C = C_1' \wedge C_2$ alors si $Solve(C) \neq$ **False** alors
$\tau = \tau_1' * \tau_2$, $\varphi = \varphi_2 \circ \varphi_1$ et $V = V_2$.

Let : si e est **let** $x = e_1$ **in** e_2 alors étant donnés
$([\tau_1/C_1], \varphi_1, V_1) = W(E, e_1, V)$ et
$([\tau_2/C_2], \varphi_2, V_2) = W(\varphi_1(E) + \{x : Gen([\tau_1/C_1], \varphi_1(E))\}, e_2, V_1)$ et
$C_1' = \varphi_2([\tau_1/C_1])$ alors
soit $C = C_1' \wedge C_2 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1'))$, si $Solve(C) \neq$ **False** alors
$\tau = \tau_2$, et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$.

Autres cas : Tous les autres cas sont des erreurs, en particulier lorsque mgu échoue ou lorsque $Solve(C) =$ **False**.

Remarque : Dans l'implantation, afin de simplifier la lecture des contraintes pour les utilisateurs nous appliquons une fonction $Simple$ sur les contraintes. Celle ci n'est pour l'instant que l'application de règles simples sur les contraintes comme $(C \Rightarrow D) \wedge (C \Rightarrow E) = C \Rightarrow (D \wedge E)$, $(C \Rightarrow D) \wedge C = C \wedge D$ ou la règle de commutativitée de $\wedge$.

6.1.3. Exemples

Expression typable Considérons l'expression e suivante (fonction `replicate` présentée avant) :

```
fun x -> mkpar(fun i -> x)
```

On a alors :

$$\begin{array}{rcl}
W(\{x:\alpha, i:\gamma\}, x, V \setminus \{\alpha,\beta,\gamma\}) & = & (\alpha, id, V \setminus \{\alpha,\beta,\gamma\}) \\
W(\{x:\alpha\}, \mathbf{fun}\ i \to x, V \setminus \{\alpha,\beta\}) & = & ([\gamma \to \alpha/\mathcal{L}(\gamma) \Rightarrow \mathcal{L}(\alpha)], id, V \setminus \{\alpha,\beta,\gamma\}) \\
W(\{x:\alpha\}, mkpar, V \setminus \{\alpha\}) & = & ([(int \to \beta) \to (\beta\ par)/\mathcal{L}(\beta)], id, V \setminus \{\alpha,\beta\}) \\
W(\{x:\alpha\}, mkpar(\mathbf{fun}\ i \to x), V \setminus \{\alpha\}) & = & ([\alpha/\mathcal{L}(\alpha)], [\beta \leftarrow \alpha, \gamma \leftarrow int], V \setminus \{\alpha,\beta,\gamma\}) \\
W(\emptyset, e, V) & = & ([\alpha \to (\alpha\ par)/\mathcal{L}(\alpha)], [\beta \leftarrow \alpha, \gamma \leftarrow int], V \setminus \{\alpha,\beta,\gamma\})
\end{array}$$

Expressions non typables

1. Considérons l'expression suivante dans laquelle deux vecteurs parallèles sont imbriqués et où `replicate` désigne l'expression précédente : `replicate(replicate 2)`.
   On obtient alors :

$$\begin{array}{rcl} W(\emptyset, 2, V \setminus \{\alpha,\beta\}) & = & (int, id, V \setminus \{\alpha,\beta\}) \\ W(\emptyset, replicate, V \setminus \{\alpha,\beta\}) & = & ([\beta \to (\beta\ par)/\mathcal{L}(\beta)], id, V \setminus \{\alpha,\beta\}) \\ W(\emptyset, replicate\ 2, V \setminus \{\alpha,\beta\}) & = & ((int\ par), [\beta \leftarrow int], V \setminus \{\alpha,\beta\}) \\ W(\emptyset, replicate, V \setminus \{\alpha\}) & = & ([\alpha \to (\alpha\ par)/\mathcal{L}(\alpha)], id, V \setminus \{\alpha,\beta\}) \\ W(\emptyset, e, V) & = & \text{error} \end{array}$$

   Une erreur apparait car $\mu = [\alpha \leftarrow (int\ par)]$ avec la contrainte $\mathcal{L}(\alpha)$ qui est résolue en **False**.

2. L'expression suivante illustre une imbrication non visible dans le type ML habituel :

```
(fun x -> fun y -> x) 1 (mkpar(fun i->4))
```

   L'implantation réalisée de notre algorithme donne la sortie suivante (avec l'option "`verbose`"):

```
         mini-BSML version 0.1-alpha
( ( (fun x -> fun y -> x) 1) (mkpar (fun i->4)));;
 VAR: x : a1
 FUN: (fun y -> x) : [(a2 -> a1) | (L(a1) => L(a2))]
 FUN: (fun x -> (fun y -> x)) : [(a1 -> (a2 -> a1)) | (L(a1) => L(a2))]
 CONST_INT: 1 : int
 APP: ((fun x -> (fun y -> x)) 1) : [(a2 -> int) | L(a2)]
 OP: mkpar : [((int -> a5) -> (a5 par)) | L(a5)]
 CONST_INT: 4 : int
 FUN: (fun i -> 4) : [(a6 -> int) | L(a6)]
 APP: (mkpar (fun i -> 4)) : (int par)
 A constraint is solved to false
```

6.2. Correction et complétude

[6] présente les preuves de correction et de complétude de notre algorithme d'inférence :

Théorème 1 *Soit une expression e, un environnement E et un ensemble V de variables de types. Si $W(e, E, V) = ([\tau/C], \varphi, V')$ alors $\varphi(E) \vdash e : [\tau/C]$*

Théorème 2 *Soit une expression e, un environnement E et V un ensemble infini de variables de types tels que $V \cap \mathcal{F}(E) = \emptyset$. S'il existe un schéma de type $[\tau_n/C_n]$ et une substitution φ' tels que $\varphi'(E) \vdash e : [\tau_n/C_n]$ alors $([\tau/C], \varphi, V') = W(e, E, V)$ est défini et il existe une substitution ψ telle que*

$$[\tau_n/C_n] = \psi([\tau/C]), \quad \textit{et} \quad \varphi' = \psi \circ \varphi \textit{ en dehors de } V$$

7. Conclusions

Le langage Bulk Synchronous Parallel ML (BSML) est un langage pour la programmation parallèle BSP en mode direct. Afin de pouvoir bénéficier d'un modèle de coût compositionnel dérivé du modèle BSP, l'imbrication de vecteurs parallèles est interdite. Des travaux antérieurs sur le langage parallèle

Caml Flight [8, 3], présentaient un mécanisme *dynamique* empêchant l'imbrication de la structure de contrôle parallèle globale `sync`. [22] a présentée une solution statique pour empêcher l'emboîtements de `sync` en utilisant le typage des effets de [20]. Notons qu'en Caml Flight le parallélisme était réellement un effet tandis que le BSλ-calcul est indépendant de la stratégie d'évaluation.

Une implantation de l'algorithme a été réalisée. Elle peut être utilisée de concert avec la bibliothèque de programmation parallèle BSMLLIB. Ce système de type a également été prouvé correct par rapport à la sémantique dynamique de ce sous-ensemble de BSML.

Nous avons étudié l'extension de ce système de types aux n-uplets et aux types sommes. Cette extension n'a pas encore été prouvée correcte par rapport à la sémantique dynamique ni incluse dans l'algorithme d'inférence. De plus, la solution trouvée pour l'instant pour les types sommes paraît écarter un nombre trop important d'expressions dont l'évaluation ne poserait pas de problèmes.

Un travail en cours concerne l'ajout de traits impératifs à BSML. Des sémantiques dynamiques ont été conçues [5]. La sûreté ne peut être assurée que par :

- soit la vérification qu'une référence créée en dehors d'un `mkpar` contient bien les mêmes valeurs sur tous les processeurs lors d'une affectation – dans ce cas des communications et une synchronisation sont nécessaires à chaque affectation d'une telle référence –

- soit par l'ajout d'une information aux références indiquant si la référence a été créée en dehors d'un `mkpar`. Cette information peut être alors utilisée dynamiquement lors de l'affectation qui est alors interdite dans le contexte d'un `mkpar`.

Nous travaillons actuellement sur le typage des effets [20] afin d'éliminer statiquement les expressions pouvant conduire à ce genre de problème.

Remerciements

Ce travail a été financé par le projet CARAML qui fait partie de l'action concertée incitative “Globalisation des ressources informatiques et des données” (ACI Grid) du Ministère de la Recherche.

Références

[1] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming: BSML and BSλ. In G. Michaelson and Ph. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect Books, 2000.

[2] Damas and Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages. Annual ACM SIGACT–SIGPLAN Symposium*, pages 207–212. Association for Computing Machinery, New York, NY, 1982.

[3] C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[4] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual Symposium on Theory of Computing (STOC)*. ACM, May 1978.

[5] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In *Submitted for publication*, 2002.

[6] Frédéric Gava. A Polymorphic Type System for BSML. Technical Report 2002-12, Master Thesis, University of Paris 12, 2002.

[7] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[8] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 56–67, Munich, June 1993. Springer.

[9] J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[10] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998.

[11] Xavier Leroy. The Objective Caml System 3.06, 2002. Web pages at `http://www.caml.org`.

[12] F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.

[13] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2002.

[14] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14^{th} *IASTED International Conference on Parallel and Distributed Computing Systems*, Cambridge, USA, November 2002.

[15] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[16] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[17] A. Merlin and G. Hains. La Machine Abstraite Catégorique BSP. In *Journées Francophones des Langages Applicatifs*. INRIA, 2002.

[18] A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, august 2001.

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

[20] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, June 1994.

[21] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[22] Julie Vachon. Une analyse statique pour le contrôle des effets de bord en Caml_Flight. JFLA 95, Journées Francophones des langages applicatifs, INRIA/CNAM, Janvier 1995

[23] Martin Odersky, Martin Sulzmann, Martin Wehr Type Inference with Constrained Types Theory and Practice of Object Systems 5(1), 1999