A Polymorphic Type System for Bulk Synchronous Parallel ML

Frédéric Gava and Frédéric Loulergue

Laboratory of Algorithms, Complexity and Logic – University Paris XII
61, avenue du général de Gaulle – 94010 Créteil cedex – France
Tel: +33 1 45 17 16 50 – Fax: +33 1 45 17 66 01
{gava,loulergue}@univ-paris12.fr

Abstract. The BSMLlib library is a library for Bulk Synchronous Parallel (BSP) programming with the functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a data structure named parallel vector, which is given by intention. In order to have an execution that follows the BSP model, and to have a simple cost model, nesting of parallel vectors is not allowed. The novelty of this paper is a type system which prevents such nesting. This system is correct w.r.t. the dynamic semantics which is also presented.

1 Introduction

Bulk Synchronous Parallel ML or BSML is an extension of the ML family of functional programming languages for programming direct-mode parallel Bulk Synchronous Parallel algorithms as functional programs. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [17] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP algorithm is said to be in *direct mode* [2] when its physical process structure is made explicit. Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [7]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

The BSMLlib library implements the BSML primitives using Objective Caml [13] and MPI [15]. It is efficient [6] and its performance follows curves predicted by the BSP cost model.

Our goal is to provide a certified programming environment for bulk synchronous parallelism. This environment will contain a byte-code compiler for BSML and an extension to the Coq Proof Assistant used to certify BSML programs. A first parallel abstract machine for the execution of BSML programs has be designed and proved correct w.r.t. the BSλ-calculus, using an intermediate semantics [5].

One of the advantages of the Objective Caml language (and more generally of the ML family of languages, for e.g. [9]) is its static polymorphic type inference [10]. In order to have both simple implementation and cost model that follows the BSP model, nesting of parallel vectors is not allowed. BSMLlib being a library, the programmer is responsible for this absence of nesting. This breaks the safety of our environment.

The novelty of this paper is a type system which prevents such nesting (section 4). This system is correct w.r.t. the dynamic semantics which is presented in section 3. We first present the BSP model, give an informal presentation of BSML (2), and explain in detail why nesting of parallel vectors must be avoided (2.1).

2 Functional Bulk Synchronous Parallelism

Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [17, 14] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization and the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time gh for any arity h.

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next super-step. The execution time of a super-step s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:$\mathrm{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$ where $w_i^{(s)} =$ local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s. The execution time $\sum_s \mathrm{Time}(s)$ of a BSP program composed of S super-steps is therefore a sum of 3 terms:$W + H * g + S * l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n, or of more complex parameters like data skew and histogram sizes.

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation as a library for Objective Caml. The so-called BSMLlib library is based on the following elements.

It gives access to the BSP parameters of the underling architecture. In particular, it offers the function `bsp_p:unit->int` such that the value of `bsp_p()` is p, the static number of processes of the parallel machine. This value does not change during execution.

There is also an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our type system enforces this restriction.

The BSML parallel constructs operates on parallel vectors. Those parallel vectors are created by: `mkpar: (int -> 'a) -> 'a par` so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. We usually write `f` as `fun pid->e` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step).

Asynchronous phases are programmed with `mkpar` and with:

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

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. Neither the implementation of BSMLlib, nor its semantics prescribe a synchronization barrier between two successive uses of `apply`.

Readers familiar with BSPlib will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by:

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

where `'a option` is defined by: `type 'a option = None | Some of 'a`.

Consider the expression: `put(mkpar(fun i->fs`$_i$`))` $(*)$

To send a value `v` from process `j` to process `i`, the function `fs`$_j$ at process `j` must be such that `(fs`$_j$` i)` evaluates to `Some v`. To send no value from process `j` to process `i`, `(fs`$_j$` i)` must evaluate to `None`.

Expression $(*)$ evaluates to a parallel vector containing a function `fd`$_i$ of delivered messages on every process. At process `i`, `(fd`$_i$` j)` evaluates to `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`.

The full language would also contain a synchronous conditional operation:

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

such that `ifat(v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core BSMLlib contains the function: `at:bool par -> int -> bool` to be used only in the construction: `if (at vec pid) then... else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases. Without it, the global control cannot take into account data computed locally.

2.1 Motivations

In this section, we present why we want to avoid nesting of parallel vectors in our language. Let consider the following BSML program:

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

This program uses the following functions:

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

`bcast 2 vec` broadcasts the value of the parallel vector `vec` held at process `2` to all other processes. The BSP cost for a call to this program is:

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

where s is the size of the value held at process `2`. Consider now the expression:

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

Its type is `(τ par) par` where τ is the type of the components of the parallel vector `vec`. A first problem is the meaning of this expression. In section 2, we said that `(mkpar f)` evaluates to a parallel vector such that process `i` holds value `(f i)`. In the case of our example, it means that process `0` should hold the value of `(bcast 0 vec)`. BSML being based on the confluent calculus [7], it is possible to evaluate `(bcast 0 vec)` sequentially. But in this case the execution time will not follow the formula (1). The cost of an expression will then depend on its context. The cost model will no more be compositional.

We could also choose that process `0` broadcasts the expression `(bcast 0 vec)` and that all processes evaluate it. In this case the execution time will follow the formula (1). But the broadcast of the expression will need communications and synchronization. This preliminary broadcast is not needed if `(bcast 0 vec)` is not under a `mkpar`. Thus we have additional costs that make the cost model still non compositional. Furthermore, this solution would imply the use of a scheduler and would make the cost formulas very difficult to write.

To avoid those problems, nesting of parallel vectors is not allowed. The typing ML programs is well-known [10] but is not suited for our language. Moreover, it is not sufficient to detect nesting of abstract type `'a par` such as the previous example. Consider the following program:

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

Its type is `int par` but its evaluation will lead to the evaluation of the parallel vector `this` inside the outmost parallel vector. Thus we have a nesting of parallel vectors which cannot be seen in the type.

Other problems arise with polymorphic values. The most simple example is a projection: `let fst = fun (a,b) -> a`. Its type is of course `'a * 'b -> 'a`. The problem is that some instantiations are incorrect. We give four cases of the application of **fst** to different kinds of values:

1. two usual values: `fst(1,2)`
2. two parallel values: `fst (mkpar(fun i -> i),mkpar(fun i -> i))`
3. parallel and usual: `fst (mkpar(fun i -> i),1)`
4. usual and parallel: `fst (1, mkpar(fun i -> i))`

The problem arises with the fourth case. Its type given by the Objective Caml system is `int`. But the evaluation of the expression needs the evaluation of a parallel vector. Thus we may be in a situation such as in `example2`. One solution would be to have a syntactic distinction between global and local variables (as in the BSλ-calculus). The type system would be simpler but it would be very inconvenient for the programmer since he would have for example to write three different versions of the `fst` function (the fourth is incorrect).

The nesting can be more difficult to detect:

```
let vec1 = mkpar(fun pid -> pid)
and vec2 = put(mkpar(fun pid -> fun from -> 1+from)) in
 let c1=(vec1,1)   and   c2=(vec2,2) in
   mkpar(fun pid ->if pid<(nproc/2) then snd c1 else snd c2)
```

The evaluation of this expression would imply the evaluation of `vec1` on the first half of the network and `vec2` on the second. But `put` implies a synchronization barrier and not `mkpar` so this will lead to mismatched barriers and the behavior of the program will be unpredictable. The goal of our type system is to reject such expressions. We are first going to equip the language with a *dynamic semantics*, then we will give the inference rules of the *static semantics* and some typing examples.

3 Dynamic Semantics of BSML

$$\begin{array}{llll}
+(n_1,n_2) & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & n \text{ with } n = n_1 + n_2 & (\delta_+) \\
\mathbf{fst}(\tilde{v}_1,\tilde{v}_2) & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & \tilde{v}_1 & (\delta_{fst}) \\
\mathbf{fix}(\mathbf{fun}\ x \to\ \tilde{e}) & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & \tilde{e}[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \to\ \tilde{e})] & (\delta_{fix}) \\
\mathbf{if\ true\ then}\ \tilde{e}_1\ \mathbf{else}\ \tilde{e}_2 & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & \tilde{e}_1 & (\delta_{ifthenelseT}) \\
\mathbf{if\ false\ then}\ \tilde{e}_1\ \mathbf{else}\ \tilde{e}_2 & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & \tilde{e}_2 & (\delta_{ifthenelseF}) \\
\mathbf{isnc}(v) & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & \mathbf{false} \text{ if } v \neq \mathbf{nc()} & (\delta_{isnc}) \\
\mathbf{isnc}(\mathbf{nc()}) & \overset{\varepsilon}{\underset{\delta}{\rightharpoonup}} & \mathbf{true} & (\delta_{isnc})
\end{array}$$

Fig. 1. δ-rules for some primitives operators

Definition of mini-BSML Reasoning on the complete definition of a functional and parallel language such as BSML, would have been complex and tedious. In order to

$$\begin{array}{lll}
\textbf{mkpar}(\textbf{fun}\ x \to e) & \xrightarrow[\delta_g]{\varepsilon} \langle e[x \leftarrow 0], \ldots, e[x \leftarrow (p-1)]\rangle & (\delta_{mkpar}) \\
\textbf{apply}(\langle \textbf{fun}\ x \to e_0, \ldots, \textbf{fun}\ x \to e_{p-1}\rangle, & & \\
\qquad \langle v_0, \ldots, v_{p-1}\rangle) & \xrightarrow[\delta_g]{\varepsilon} \langle e_0[x \leftarrow v_0], \ldots, e_{p-1}[x \leftarrow v_{p-1}]\rangle & (\delta_{apply}) \\
\textbf{if}\ \langle \ldots, \overbrace{\textbf{true}}^{n}, \ldots\rangle\ \textbf{at}\ v_g\ \textbf{then}\ e_{g1}\ \textbf{else}\ e_{g2} & \xrightarrow[\delta]{\varepsilon} e_{g1} \quad \text{if } v_g = n & (\delta_{ifatT}) \\
\textbf{if}\ \langle \ldots, \overbrace{\textbf{false}}^{n}, \ldots\rangle\ \textbf{at}\ v_g\ \textbf{then}\ e_{g1}\ \textbf{else}\ e_{g2} & \xrightarrow[\delta]{\varepsilon} e_{g2} \quad \text{if } v_g = n & (\delta_{ifatF}) \\
\textbf{put}(\langle \textbf{fun}\ dst \to e_0, \ldots, \textbf{fun}\ dst \to e_{p-1}\rangle) & \xrightarrow[\delta_g]{\varepsilon} \langle e'_0, \ldots, e'_{p-1}\rangle & (\delta_{put})
\end{array}$$

where $\forall i\ e'_j = \textbf{let}\ v^i_0 = e_0[dst \leftarrow i]\ \textbf{in} \ldots \textbf{let}\ v^i_{p-1} = e_{p-1}[dst \leftarrow i]\ \textbf{in}\ f_i$
where $\forall i \forall j\ v^i_j \notin F\ (e_j)$
and where $f_i = \textbf{fun}\ x \to \textbf{if}\ x = 0\ \textbf{then}\ v^i_0\ \textbf{else} \ldots \textbf{if}\ x = (p-1)\ \textbf{then}\ v^i_{p-1}\ \textbf{else nc()}$

Fig. 2. δ-rules for some parallel operators

$$\begin{array}{rlll|lll}
e ::= & x & \text{variable} & & c & \text{constant} \\
\mid & op & \text{primitive operation} & & \textbf{fun}\ x \to e & \text{function abstraction} \\
\mid & (e\ e) & \text{application} & & \textbf{let}\ x = e\ \textbf{in}\ e & \text{local binding} \\
\mid & (e, e) & \text{pair} & & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional} \\
\mid & \multicolumn{3}{l}{\textbf{if}\ e\ \textbf{at}\ e\ \textbf{then}\ e\ \textbf{else}\ e \ \ \text{global conditional}} & &
\end{array}$$

Fig. 3. mini-Bsml syntax

Local values:

$$\begin{array}{rll}
v ::= & \textbf{fun}\ x \to e & \text{functional value} \\
\mid & c & \text{constant} \\
\mid & op & \text{primitive} \\
\mid & (v, v) & \text{pair}
\end{array}$$

Global values:

$$\begin{array}{rll}
v_g ::= & \textbf{fun}\ x \to e_g & \text{functional value} \\
\mid & c & \text{constant} \\
\mid & op & \text{primitive} \\
\mid & (v_g, v_g) & \text{pair} \\
\mid & \langle v, \ldots, v\rangle & \text{p-wide parallel vector}
\end{array}$$

Fig. 4. Values

simplify the presentation and to ease the formal reasoning, this section introduces a core language. It is an attempt to trade between integrating the principal features of functional and BSP language, and being simple. The *expressions* of mini-BSML, written e possibly with a prime or subscript, have the abstract syntax given in Figure 3. In this grammar, x ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e, to an argument e'. The form $\textbf{fun}\ x \rightarrow e$ is the so-called and well-known lambda-abstraction that defines the first-class function whose parameter is x and whose result is the value of e. Constants c are the integers 1, 2, the booleans and we assume having a unique value **()** that have the type unit.

The set of primitive operations op contains arithmetic operations, fix-point operator **fix**, test function **isnc** of the **nc** constructor (which plays the role of the `None` constructor in Objective Caml) and our parallel operations: **mkpar**, **apply**, **put** and **ifat**.

Before typing these expressions, we present the dynamic semantics of the language, i.e., how the expressions of mini-BSML are computed to *values*. There is one semantics per value of p, the number of processes of the parallel machine. In the following, $\forall i$ means $\forall i \in \{0,\ldots,p-1\}$. There is two kinds of values: local and global values (Figure 4). e_g are expressions extended with parallel vectors of expressions : $\langle e,\ldots,e\rangle$. We noted $\tilde{v}$ (resp. $\tilde{e}$) for a local or global value (resp. expressions or extended expressions).

Small-step semantics The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we use a a *small-step semantics*. It consists of a predicate between extended expressions and another extended expression defined by a set of axioms and rules called steps. The small-step semantics describes all the steps of the calculus from an extended expression to a global value and has the following form:

$$\begin{array}{ll} e_g \rightharpoonup e'_g & \text{for one step} \\ e_{g_0} \rightharpoonup e_{g_1} \rightharpoonup \ldots \rightharpoonup v_g & \text{for all the steps of the calculus} \end{array}$$

We note $\overset{*}{\rightharpoonup}$, for the transitive closure of $\rightharpoonup$ and note $e_{g_0} \overset{*}{\rightharpoonup} v_g$ for $e_{g_0} \rightharpoonup e_{g_1} \rightharpoonup \ldots \rightharpoonup v_g$. To define the relation, $\rightharpoonup$, we begin with some axioms for two relations, $\overset{\varepsilon}{\rightharpoonup}$ and $\underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}}$, of the head reduction:

$$\begin{array}{cc} (\textbf{fun}\ x \rightarrow e)\ v \overset{\varepsilon}{\rightharpoonup} e[x \leftarrow v] & (\textbf{let}\ x = v\ \textbf{in}\ e) \overset{\varepsilon}{\rightharpoonup} e[x \leftarrow v] \\ (\textbf{fun}\ x \rightarrow e_g)\ v_g \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} e_g[x \leftarrow v_g] & (\textbf{let}\ x = v_g\ \textbf{in}\ e_g) \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} e_g[x \leftarrow v_g] \end{array}$$

We write $e[x \leftarrow v]$ (resp. $e_g[x \leftarrow v_g]$) the expression by substituting all the free occurrences of x in e by v (resp. extended expression). For the primitive operators we have some axioms, the δ-rules, noted $\underset{\delta}{\overset{\varepsilon}{\rightharpoonup}}$ (Figure 1 for global and local values) and in the same manner we noted $\underset{\delta_g}{\overset{\varepsilon}{\rightharpoonup}}$ for the δ-rules of the parallel operators (Figure 2).

We define two kinds of *head reductions*:

Local reduction: $\underset{l}{\overset{\varepsilon}{\rightharpoonup}} = \overset{\varepsilon}{\rightharpoonup} \cup \underset{\delta}{\overset{\varepsilon}{\rightharpoonup}}$ Global reduction: $\underset{g}{\overset{\varepsilon}{\rightharpoonup}} = \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} \cup \underset{\delta}{\overset{\varepsilon}{\rightharpoonup}} \cup \underset{\delta_g}{\overset{\varepsilon}{\rightharpoonup}}$

It is easy to see that we cannot always make a head reduction. We have to reduce in depth in the extended sub-expression. To define this deep reduction, we use the following inference rules:

$$\frac{e \overset{\varepsilon}{\underset{l}{\rightharpoonup}} e'}{\Gamma_l(e) \rightharpoonup \Gamma_l(e')} \qquad\qquad \frac{e_g \overset{\varepsilon}{\underset{g}{\rightharpoonup}} e'_g}{\Gamma(e_g) \rightharpoonup \Gamma(e'_g)}$$

In this rule, Γ and Γ_l are *evaluation contexts*, i.e., an expression with a hole and has the abstract syntax given in figure 5. With the evaluation context Γ_l, we can remark that the head reduction is always in a component of a parallel vector (and not for Γ), i.e, a local evaluation. Thus our two kinds of contexts exclude each other by construction.

$$\begin{array}{rcll}
\Gamma & ::= & [] & \text{head evaluation} \\
& \mid & (\Gamma\ \tilde{e}) & \text{right application evaluation} \\
& \mid & (\tilde{v}\ \Gamma) & \text{left application evaluation} \\
& \mid & (\Gamma, \tilde{e}) & \text{left pair evaluation} \\
& \mid & (\tilde{v}, \Gamma) & \text{right pair evaluation} \\
& \mid & \textbf{let}\ x = \Gamma\ \textbf{in}\ \tilde{e} & \text{let evaluation} \\
& \mid & \textbf{if}\ \Gamma\ \textbf{then}\ \tilde{e_1}\ \textbf{else}\ \tilde{e_2} & \text{conditional} \\
& \mid & \textbf{if}\ \Gamma\ \textbf{at}\ e_{g_1}\ \textbf{then}\ e_{g_2}\ \textbf{else}\ e_{g_3} & \text{global conditional} \\
& \mid & \textbf{if}\ v_g\ \textbf{at}\ \Gamma\ \textbf{then}\ e_{g_2}\ \textbf{else}\ e_{g_3} & \text{global conditional} \\
\\
\Gamma_l & ::= & (\Gamma_l\ e_g) & \\
& \mid & (v_g\ \Gamma_l) & \\
& \mid & (\Gamma_l, e_g) & \\
& \mid & (v_g, \Gamma_l) & \\
& \mid & \textbf{let}\ x = \Gamma_l\ \textbf{in}\ e_g & \\
& \mid & \textbf{if}\ \Gamma\ \textbf{then}\ e_{g_1}\ \textbf{else}\ e_{g_2} & \\
& \mid & \textbf{if}\ \Gamma\ \textbf{at}\ e_{g_1}\ \textbf{then}\ e_{g_2}\ \textbf{else}\ e_{g_3} & \\
& \mid & \textbf{if}\ v_g\ \textbf{at}\ \Gamma\ \textbf{then}\ e_{g_2}\ \textbf{else}\ e_{g_3} & \\
& \mid & \langle \Gamma, e_1, \ldots, e_{p-1} \rangle & \text{parallel vector, first component} \\
& \vdots & & \\
& \mid & \langle e_0, \ldots, \overbrace{\Gamma}^{i}, e_{i+1}, \ldots, e_{p-1} \rangle & \text{i}^{\text{th}}\text{ component} \\
& \vdots & & \\
& \mid & \langle e_0, \ldots, e_{p-2}, \Gamma \rangle & \text{last component}
\end{array}$$

Fig. 5. Evaluation contexts

4 A Polymorphic Type System

Type algebra We begin by defining the term algebra for the basic kinds of semantic objects: the simple types. Simple types are defined by the following grammar:

$\tau ::=$	κ	base type (bool, int, unit etc.)
$\mid$	α	type variable
$\mid$	$\tau_1 \to \tau_2$	type of function from τ_1 to τ_2
$\mid$	$\tau_1 * \tau_2$	type for pair
$\mid$	$(\tau\ par)$	parallel vector type

We want to distinguish between three subsets of simple types. The set of local types L, which represent usual Objective Caml types, the variable types V for polymorphic types and global types G, for parallel objects. The local types (written $\dot{\tau}$) are:

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

the variable types are (written $\breve{\tau}$):

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

and the global types (written $\overline{\tau}$) are:

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

Of course, we have $L \cap G = \emptyset$ and $V \cap G = \emptyset$. But, it is easy to see that not every instantiations from a variable type are in one of these kinds of types. Take for example, the simple type $(\alpha\ par) \to int$ or the simple type $(\alpha\ par)$ and the instantiation $\alpha = (int\ par)$: it leads to a nesting of parallel vectors.

To remedy this problem, we will use *constraints* to say which variables are in L or not. For a polymorphic type system, with this kind of constraints, we introduce a *type scheme* with constraints to generically represent the different types of an expression:

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

Where τ is a simple type and C is a constraint of classical propositional calculus given by the following grammar:

$C ::=$	**True**	true constant constraints
$\mid$	**False**	false constant
$\mid$	$L(\alpha)$	locality of variable of type
$\mid$	$C_1 \wedge C_2$	conjonction of two constraints
$\mid$	$C_1 \Rightarrow C_2$	implication of two constraints

When the set of variables is empty, we simply write $[\tau / C]$ and do not write the constraints when they are equal to **True**. We suppose that we work modulo these following equations that are natural for the $\wedge$ operators: **True** $\wedge\ C = C$, $C \wedge C = C$ and the commutativity of the $\wedge$ operator.

For a simple type τ, $L(\tau)$ says that the simple type is in L and we uses the following inductive rules to not have the locality of a type but of its variables:

$$\begin{array}{llll} L(\alpha) & = \textbf{True} \quad \text{if } \alpha \in \kappa & L(\tau\ par) & = \textbf{False} \\ L(\tau_1 \to \tau_2) & = L(\tau_1) \wedge L(\tau_2) & L(\tau_1 * \tau_2) & = L(\tau_1) \wedge L(\tau_2) \end{array}$$

In the type system and for the substitution of a type scheme we will use rules to construct constraints from a simple type called *basic constraints*. We note C_τ for the basic constraints from the simple type τ and we use the following inductive rules:

$$\begin{array}{llll} C_\tau & = \textbf{True} \quad \text{if } \tau \text{ atomic} & C_{(\tau_1 \to \tau_2)} & = C_{\tau_1} \wedge C_{\tau_2} \wedge L(\tau_2) \Rightarrow L(\tau_1) \\ C_{(\tau\ par)} & = L(\tau) \wedge C_\tau & C_{(\tau_1 * \tau_2)} & = C_{\tau_1} \wedge C_{\tau_2} \end{array}$$

In our type system, we will use basic constraints and constraints associated to sub-expressions that are needed in cases similar to `example2`.

The set of free variable of a type scheme is defined by:

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

where the free variables of the type and the constraints are defined by trivial structural induction. We note *Dom* for the domain of a *substitution* (i.e. a finite application of variables of type to simple types). With these definitions we can define the substitution on a type scheme:

Definition 1 *The substitution on a type scheme is defined by:*

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

if $\alpha_1 \ldots \alpha_n$ *are out of reach of* φ*.*

We say that a variable α is *out of reach* of a substitution φ if: $\varphi(\alpha) = \alpha$, i.e φ don't modify α (or α is not in the domain of φ) and if α is not free in $[\tau/C]$, then α is not free in $\varphi([\tau/C])$, i.e, φ do not introduce α in its result. The condition that $\alpha_1 \ldots \alpha_n$ are out of reach of φ can always be validated by renaming first $\alpha_1 \ldots \alpha_n$ with fresh variables (we suppose that we have an infinite set of variables). The substitution on the simple type and of the constraints are defined by trivial structural induction.

Instantiation and Generalization A type scheme can be seen like the set of types given by instantiation of the quantifier variables. We introduce the notion of instance of a type scheme with constraints.

Definition 2 *We note* $[\tau/C] \leq \forall \alpha_1 ... \alpha_n.[\tau'/C']$ *if and only if, there exists a substitution* φ *of domain* $\alpha_1, \ldots, \alpha_n$ *where* $[\tau/C] = \varphi([\tau'/C'])$.

We write E for an *environment* which associates type schemes to free variables of an expression. It is an application from free variables (identifiers) of expressions to type scheme. We note $Dom(E) = \{x_1, \ldots, x_n\}$ for its domain, i.e the set of variables

$$
\begin{array}{llll}
TC(i) & = \mathbf{int} \quad i = 0,1,\ldots & TC(\mathbf{fix}) & = \forall\alpha.(\alpha \to \alpha) \to \alpha \\
TC(b) & = \mathbf{bool} \quad b = \mathbf{true},\mathbf{false} & TC(\mathbf{fst}) & = \forall\alpha\beta.[(\alpha * \beta) \to \alpha / L(\alpha) \Rightarrow L(\beta)] \\
TC(()) & = \mathbf{unit} & TC(\mathbf{snd}) & = \forall\alpha\beta.[(\alpha * \beta) \to \beta / L(\beta) \Rightarrow L(\alpha)] \\
TC(+) & = (\mathbf{int} * \mathbf{int}) \to \mathbf{int} & TC(\mathbf{mkpar}) & = \forall\alpha.[(\mathbf{int} \to \alpha) \to (\alpha\ par)/L(\alpha)] \\
TC(\mathbf{nc}) & = \forall\alpha.\mathbf{unit} \to \alpha & TC(\mathbf{isnc}) & = \forall\alpha.[\alpha \to \mathbf{bool}/L(\alpha)]
\end{array}
$$

$$
\begin{array}{ll}
TC(\mathbf{apply}) & = \forall\alpha\beta.[((\alpha \to \beta)\ par * (\alpha\ par)) \to (\beta\ par)/L(\alpha) \wedge L(\beta)] \\
TC(\mathbf{put}) & = \forall\alpha.[(\mathbf{int} \to \alpha)\ par \to (\mathbf{int} \to \alpha)\ par/L(\alpha)]
\end{array}
$$

Fig. 6. Definition of *TC*

associated. We assume that all the identifiers are distinct. The empty mapping is written $\emptyset$ and $E(x)$ for the type scheme associated with x in E. The substitution φ on E is a point to point substitution on the domain of E. The set of free variables is naturally defined on the free variables on all the type scheme associated in the domain of E.

Finally, we write $E + \{x : \sigma\}$ for the extension of E to the mapping of x to σ. If, before this operation, we have $x \in Dom(E)$, we can replace the range by the new type scheme for x. To continue with the introduction of the type system, we define how to generalize a type scheme. Yet, type schemes have universal quantified variables, but not all the variables of a type scheme can.

Definition 3 *Given an environment E, a type scheme $[\tau/C]$ without universal quantification, we define an operator Gen to introduce universal quantification: $Gen([\tau/C],E) = \forall\alpha_1...\alpha_n.[\tau/C]$ where $\{\alpha_1,...,\alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E)$*

With this definition, we have introduced polymorphism. The universal quantification gives the choice for the system to take the good type from a type scheme.

Inductive rules We note TC (Figure 6) the function which associates a type scheme to the constants and to the primitive operations.

We formulate type inference by a deductive proof system that assigns a type to an expression of the language. The context in which an expression is associated with a type is represented by an environment which maps type scheme to identifiers. Deductions produce conclusions of the form $E \vdash e : [\tau/C]$ which are called *typing judgments*, they could be read as: "in the type environment E, the expression e has the type $[\tau/C]$". The static semantics manipulates type schemes by using the mechanism of generalization and instantiation specified in the previous sections. Now the inductive rules of the type system are given in the Figure 7.

In all the inductive rules, if a constraint C is such that $Solve(C) = \mathbf{False}$ then the inductive rule cannot be applied and then the expression is not well typed. To *Solve* the constraints we use the classical boolean reduct rules of propositional calculus and the rules to transform the locality of type to constraints. Our constraints are a sub part of the propositional calculus, so *Solve* is a decidable function.

Like traditional static type systems, the case (Op), $(Const)$ and (Var) used the definition of an instance of a type scheme. The rule (Fun), introduce a new type scheme on the environment with the basic constraints from the simple type for carrying of well

$$\frac{[\tau/C] \leq E(x)}{E \vdash x : [\tau/C]}\text{(Var)} \qquad \frac{[\tau/C] \leq TC(c)}{E \vdash c : [\tau/C]}\text{(Const)} \qquad \frac{[\tau/C] \leq TC(op)}{E \vdash op : [\tau/C]}\text{(Op)}$$

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

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

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

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

$$\frac{E \vdash e_1 : [\textbf{bool}/C_{e_1}] \quad E \vdash e_2 : [\tau/C_{e_2}] \quad E \vdash e_3 : [\tau/C_{e_3}]}{E \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : [\tau/C_{e_1} \wedge C_{e_2} \wedge C_{e_3}]}\text{(Ifthenelse)}$$

$$\frac{E \vdash e_1 : [\textbf{bool}\ par/C_{e_1}] \quad E \vdash e_2 : [\textbf{int}/C_{e_2}] \quad E \vdash e_3 : [\tau/C_{e_3}] \quad E \vdash e_4 : [\tau/C_{e_4}]}{E \vdash \textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4 : [\tau/C_{e_1} \wedge C_{e_2} \wedge C_{e_3} \wedge C_{e_4} \wedge (L(\tau) \Rightarrow \textbf{False})]}\text{(Ifat)}$$

Fig. 7. The inductive rules

parameters. In the rules (App), $(Pair)$ and (Let) we make the conjunction of the constraints to known if the two sub-cases are correct each other. Moreover, in (Let), we introduce the fact that $L(\tau_2) \Rightarrow L(\tau_1)$ because an expression like **let** $x = e_1$ **in** e_2 can be seen like (**fun** $x \to e_2$) e_1. So we have to protect our type system against expression from global values to local values like in `example 2`. The rule $(ifthenelse)$ and $(ifat)$ make also the conjunction of the constraints. The **if then else** construction could return global or usual value. But the **if at** is a synchronous construction which needs global values so we add the fact that $(L(\tau) \Rightarrow$ **False**$)$ to not allow a return usual value (i.e. τ in L).

The basic constraints are important in our type system but are not suitable. For the following example, a parallel identity:

```
fun x -> if (mkpar (fun i -> true) at 0 then x else x
```

the basic constraints are not sufficient. Indeed, the simple type given by Objective Caml is $\alpha \to \alpha$ and the basic constraints $(L(\alpha) \Rightarrow L(\alpha))$ are always solved to **True**. But it is easy to see that the variable x (of type α) could not be a usual value. Our type system, with constraints from the sub-expression (here **ifat**) would give the type scheme: $[\alpha \to \alpha/L(\alpha) \Rightarrow$ **False**$]$ (i.e, α could not be a usual value and the instantiation are in G).

Afterwards, we need to know when a constraint is *Solved* to **True**, i.e. it is always a valid constraint. It will be important, notably for the correction of the type system:

Definition 4 *We write $\varphi \models C$, if the substitution φ on the free variables of C is such that $\mathcal{F}(\varphi(C)) = \emptyset$ and $Solve(\varphi(C)) =$ **True**.*

We also write $\phi_C = \{\varphi \mid \varphi \models C\}$ for the set of all the substitutions that have these properties.

Safety To ensure safety, the type system has been proved correct with respect to the small-step semantics. We say that an extended expression e_g is in *normal form* if and only if $e_g \not\rightharpoonup$, i.e., there is no rule which could be applicate to e_g.

Theorem 1 (Typing safety) *If $\emptyset \vdash e : [\tau/C]$ and $e \overset{*}{\rightharpoonup} e'_g$ and e'_g is in normal form, then e'_g is a value v_g and there exists C' such that $\forall \varphi \in \phi_C$ then $\varphi \models C'$ and $\emptyset \vdash v_g : [\tau/C']$.*

Proof: see [1].

Why C' and not C ? Because with our type system, the constraints of a typing judgment for e contains constraints of the sub-expression of e. After evaluation, some of this sub-expression could be reduced. Example: $\mathbf{let}\ f = (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a)\ \mathbf{in}\ 1$ have the type $[int/L(\alpha) \Rightarrow L(\beta)]$. This expression reduced to 1 has the type int. Thus we have C' is less constrained than C and we do not have problem with compositionality.

Examples For the `example 2` given at the beginning of this text, the type scheme given for `this` is $(\mathbf{int}\ par)$ and the type for pid is the usual $\mathbf{int}$. So after a (Let) rule, the constraints for this let binding construction are $C = L(\mathbf{int}) \Rightarrow L(\mathbf{int}\ par)$ with $Solve(C) = \mathbf{False}$. So the expression is not well-typed (Figure 8 gives a part of the typing judgment).

$$\dfrac{\dfrac{\dfrac{\dots}{\{pid:\mathbf{int}\} \vdash \mathbf{mkpar}(\mathbf{fun}\ i \to i) : (\mathbf{int}\ par)} \qquad \dfrac{\mathbf{int} \leq \mathbf{int}}{\{pid:\mathbf{int}\} \vdash pid : \mathbf{int}}}{\{pid:\mathbf{int}\} \vdash \mathbf{let}\ this = \mathbf{mkpar}(\mathbf{fun}\ i \to i)\ \mathbf{in}\ pid) : ?}}{\emptyset \vdash (\mathbf{fun}\ pid \to \mathbf{let}\ this = \mathbf{mkpar}(\mathbf{fun}\ i \to i)\ \mathbf{in}\ pid) : ?}$$

Fig. 8. Typing judgment of a part of `example 2`

In the parallel and usual projection (see Figure 9), the expression is well-typed like we want in the previous section. In Figure 10, we present the typing judgment of another example, accepted by the type system of Objective Caml, but not by ours. For the usual and parallel projection, the projection **fst** has the simple type $(\mathbf{int} * (\mathbf{int}\ par)) \to \mathbf{int}$. But, with our type scheme substitution, the constraints of this operator are : $C = L(\mathbf{int}) \Rightarrow L(\mathbf{int}\ par)$. Effectively, we have $Solve(C) = \mathbf{False}$ and the expression is rejected by our type system. In the typing judgments given in the figures, we noted : ? when the type derivation is impossible for our type system.

$$\dfrac{\dfrac{\dots}{\emptyset \vdash \mathbf{fst} : (\mathbf{int} * \mathbf{int}\ par) \to \mathbf{int} : ?} \qquad \dfrac{\dfrac{\dots}{\emptyset \vdash : (\mathbf{mkpar}\ (\mathbf{fun}\ i \to i)) : \mathbf{int}\ par} \quad \dfrac{\mathbf{int} \leq \mathbf{int}}{\vdash 1 : \mathbf{int}}}{\emptyset \vdash (\mathbf{mkpar}\ (\mathbf{fun}\ i \to i), 1) : (\mathbf{int}\ par * \mathbf{int})}}{\emptyset \vdash \mathbf{fst}\ (\mathbf{mkpar}\ (\mathbf{fun}\ i \to i), 1) :\ \mathbf{int}\ par}$$

Fig. 9. Typing judgment of the third projection example

$$\dfrac{\dfrac{\dots}{\emptyset \vdash \mathbf{fst} : (\mathbf{int} * \mathbf{int}\ \mathit{par}) \to \mathbf{int} : ?} \quad \dfrac{\dfrac{\mathbf{int} \le \mathbf{int}}{\vdash 1 : \mathbf{int}} \quad \dfrac{\dots}{\emptyset \vdash : (\mathbf{mkpar}\ (\mathbf{fun}\ i \to i)) : \mathbf{int}\ \mathit{par}}}{\emptyset \vdash (1, \mathbf{mkpar}\ (\mathbf{fun}\ i \to i)) : (\mathbf{int} * \mathbf{int}\ \mathit{par})}}{\emptyset \vdash \mathbf{fst}\ (1, \mathbf{mkpar}\ (\mathbf{fun}\ i \to i)) : ?}$$

Fig. 10. Typing judgment of the fourth projection example

5 Related Works

In previous work on Caml Flight [3], another parallel ML, the global parallel control structure was prevented dynamically from nesting. A static analysis [16] have been designed but for some kinds of nesting only and in Caml Flight, the parallelism is a side effect while is is purely functional in BSML.

The libraries close to our framework based either on the functional language Haskell [8] or on the object-oriented language Python [4] propose flat operations similar to ours. In the latter, the programmer is responsible for the non nesting of parallel vectors. In the former, the nesting is prohibited by the use of monads. But the distinction between global and local expressions is syntactic thus less general than our framework. For example, the programmer need to write three version of `fst`. Furthermore, Haskell is a lazy language: it is less efficient and the cost prevision is difficult [12].

A general framework for type inference with constrained type called HM(X) [11] also exists and could be used for a type system with only basic constraints. We do not used this system for three reasons: (1) this type system has been proved for λ-calculus (and sequential languages whose types systems need constraints) and not for our theoretical calculus, the BSλ-calculus with its two level structure (local and global); (2) in the logical type system, the constraints depend of sub-expression are not present; (3) in our type system, our abstraction could not be valid and generate constraints (not in HM(X)). Nevertheless, the ideas (but not the framework itself) of HM(X) could be used for generalized our work for tuple, sum types and imperative features.

6 Conclusions and Future Work

The Bulk Synchronous Parallel ML allows direct mode Bulk Synchronous Parallel (BSP) programming. To preserve a compositional cost model derived form the BSP cost model, the nesting of parallel vectors is forbidden. The type system presented in this paper allows a static avoidance of nesting. Thus the pure functional subset of BSML is safe. We have also designed an algorithm for type inference and implemented it. It can be used in conjunction with the BSMLlib programming library. The extension of the type system to tuples and sum types have been investigated but not yet proved correct w.r.t. the dynamic semantics nor included in the type inference algorithm.

A further work will concern imperative features. Dynamic semantics of the interaction of imperative features with parallel operations have been designed. To ensure safety, communications may be needed in case of affectation or references may contain additional information used dynamically to insure that dereferencing of references pointing to local value will give the same value on all processes. We are currently working on the typing of effects to avoid this problem statically.

Acknowledgments This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML (www.caraml.org).

References

1. Frédéric Gava. A Polymorphic Type System for BSML. Technical Report 2002-12, University of Paris Val-de-Marne, LACL, 2002.
2. A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
3. G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.
4. K. Hinsen. Parallel Programming with BSP in Python. Technical report, Centre de Biophysique Moléculaire, 2000.
5. F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2001.
6. F. Loulergue. Implementation of a Functional BSP Programming Library. In *14th Iasted PDCS Conference*, pages 452–457. ACTA Press, 2002.
7. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
8. Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.
9. R. Milner and al. *The Definition of Standard ML*. MIT Press, 1990.
10. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
11. M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
12. C. Paraja, R. Pena, F. Rubio, and C. Segura. A functional framework for the implementation of genetic algorithms: Comparing Haskell and Standard ML. In *Trends in Functional Programming*, volume 2. Intellect Books, 2001.
13. D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.
14. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.
15. M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
16. J. Vachon. Une analyse statique pour le contrôle des effets de bords en Caml-Flight beta. In C. Queinnec et al., editors, *JFLA*, INRIA, Janvier 1995.
17. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.