Semantics of a Functional BSP Language with Imperative Features

Frédéric Gava[a] and Frédéric Loulergue[a]

[a]Laboratory of Algorithms, Complexity and Logic,
61, avenue du Général de Gaulle, 94010 Créteil cedex, France

The Bulk Synchronous Parallel ML (BSML) is a functional language for Bulk Synchronous Parallel (BSP) programming, on top of the sequential functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a parallel data structure named parallel vector, which is given by intention. The Objective Caml language is a functional language but it also offers imperative features. This paper presents formal semantics of BSML with references, assignment and dereferencing.

1. Introduction

Declarative parallel languages are needed to ease the programming of massively parallel architectures. We are exploring thoroughly the intermediate position of the paradigm of algorithmic skeletons [1,7] in order to obtain universal parallel languages whose execution costs can be easily determined from the source code (in this context, cost means the estimate of parallel execution time). This last requirement forces the use of explicit processes corresponding to the parallel machine's processors. *Bulk Synchronous Parallel* (BSP) computing [11] is a parallel programming model which uses explicit processes, offers a high degree of abstraction and yet allows portable and predictable performance on a wide variety of architectures. Our BSML [6] can be seen as an algorithmic skeletons language, because only a finite set of operations are parallel, but is different by two main points: (a) our operations are universal for BSP programming and thus allow the implementation of more classical algorithmic skeletons. It is also possible for the programmer to implement additional skeletons. Moreover performance prediction is possible and the associated cost model is the BSP cost model. Those operations are implemented as a library [5] for the functional programming language Objective Caml [8]; (b) the parallel semantics of BSML are formal ones. We have a confluent calculus, a distributed semantics and a parallel abstract machine, each semantics has been proved correct with respect to the previous one.

Our semantics are based on extension of the λ-calculus, but our `BSMLlib` library (a partial implementation of the complete BSML language) is for the Objective Caml language which contains imperative features. Imperative features can be useful to write (sequential) programs in Objective Caml (for example, many interpreters of λ-calculus written in Objective Caml use a imperative counter in order to generate fresh variable names). Sometimes imperative features are also needed to improve efficiency. Thus to offer such expressivity in BSML we have to add imperative features to BSML. In the current version of the `BSMLlib` library the use of imperative features is unsafe and may lead to runtime errors. This paper explores formal semantics of BSML with imperative features.

We first describe functional bulk synchronous parallel programming and the problems that appear with the use of imperative features (section 2). We then give two formal semantics of BSML with imperative features whose parallel executions are different (section 3) and conclude (section 4).

2. Preliminaries

We assume here some knowledge of Bulk Synchronous Parallelism (BSP) [9]. There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation: 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. The value of this variable 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 [4]. In our framework, indeterminism and deadlocks are avoided and it is possible to easily prove programs using the Coq proof assistant [2].

The BSML parallel constructs operate 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`. The expression `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`.

`put:(int->'a option)par->(int->'a option)par` allows to express communications and synchronizations, 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 as (`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 BSML language would also contain a global synchronous conditional operation. This `ifat:(bool par)*in *'a*'a->'a` operation is 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 global synchronous conditional operation can not be defined as a function. That is why the core `BSMLlib` library contains the function `at:bool par->int->bool` to be used only in the global expression: `if (at vec pid) then ...ẽlse ...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases. Global conditional is necessary of express algorithms like: **Repeat** Parallel Iteration **Until** Max of local errors $< \epsilon$. Without it, the global control cannot take into account data computed locally.

Objective Caml offers to the programmer an important extension of functional languages: imperative features. They have been added to functional languages to offer more expressiveness. Classically, this modification is added to functional languages by the possibility of *assignment* and *allocation* of a variable or a data structure.

The idea is to add *references*. A reference is a cell of the memory which could be modified by the program. One creates a reference with the allocation's **ref**(e) construction which gives a new reference in the memory initialized to the value of e. The value kept by the reference is called the *stored value*. To use and read the stored value (*dereferencing*), we need an operation, written **!**, to extract it. Finally, we can modify the content of our reference by replacing this value by another. This operation is called *assignment* and written: $e_1 := e_2$. We use the same notations as Objective Caml. A reference binding to an identifier in a functional language is like a variable in an imperative language.

Imperative features are not a trivial extension of functional language. First, the value of an expression changes with the values of the free variables. If these variables have a known value, the evaluation of the sub-expressions could be done independently. For imperative language, it is not the case (and also for imperative extensions): the evaluation of a sub-expression could modify a reference by an assignment and thus affects the evaluation of the other sub-expressions which used this reference. A secondly difficulty came from the shared references which could not pass the well-known and classical β-reduction of the functional languages. Take for example: `let r=ref 2 in r:=!r*!r;(!r+1)`.

The instances of `r` are for the reference, allocated by the `ref 2` sub-expression. If we make a natural β-reduction to our expression, we would have: `(ref 2):=!(ref 2)*!(ref 2);(!(ref 2)+1)` which allocates four different references and do not have the same behavior.

To extend the dynamic semantics of functional languages and keep out the problem of the shared allocations, *locations* (written ℓ) and *store* (written s) have been added. A store, is a partial function from locations to values and a reference is evaluated to a location. In the following, we give the reduction of expressions starting from an empty store.

First, the left sub-expression of the `let` construction is evaluated and a new location ℓ is created in the store. Second, the β-reduction can be applied and finally the right sub-expression of the `let` construction is evaluated with the classical rules of a functional language:

$$\begin{array}{lllll|lllll}
 & \textbf{let } r = \textbf{ref}(2) \textbf{ in } r :=!r*!r;(!r+1) & / & \emptyset & & \rightharpoonup & \ell := 4;!(\ell+1) & / & \{\ell \mapsto 2\} \\
\rightharpoonup & \textbf{let } r = \ell \textbf{ in } r :=!r*!r;(!r+1) & / & \{\ell \mapsto 2\} & & \rightharpoonup & (!\ell+1) & / & \{\ell \mapsto 4\} \\
\rightharpoonup & \ell :=!\ell*!\ell;(!\ell+1) & / & \{\ell \mapsto 2\} & & \rightharpoonup & (4+1) & / & \{\ell \mapsto 4\} \\
\rightharpoonup & \ell := 2*2;(!\ell+1) & / & \{\ell \mapsto 2\} & & \rightharpoonup & 5 & / & \{\ell \mapsto 4\}
\end{array}$$

2.1. BSML with imperative features

BSML is a parallel functional language based on BSP whose architecture model contains a set of processor-memory pairs and a network. Thus in the current implementation each processor can reach its own memory, and it causes problems. Take for example, the expression:

```
let a = ref(0) in let danger = mkpar(fun pid -> a:=pid; pid mod 2=0) in
if (at danger !a) then e1 else e2
```

First, this expression creates a location `a` at each processor which is initialized at 0 everywhere. For the `BSMLlib` library each processor has this value in its memory. Second, a boolean parallel vector `danger` is created which is trivially `true` if the processor number is even or `false` otherwise. Thus, from the `BSMLlib` point of view, the location `a` has now a different value at each processor. After the **ifat** construct, some processor would execute `E1` and some other `E2`. But, the **ifat** is a global synchronous operation and all the processors need to execute the same branch of the conditional. If this expression had been evaluated with the `BSMLlib` library, we would have obtained an incoherent result and a crash of the BSP machine. The goal of our new semantics is to dynamically reject this kind of problems (and to have an exception raised in the implementation).

3. Dynamic Semantics of BSML with Imperative Features

This section introduces the syntax and dynamic semantics of a core langage, together with some conventions, definitions and notations that are used in the paper.

3.1. Syntax

The expressions of mini-BSML, written e, have the following abstract syntax:

$$\begin{array}{rcl} e & ::= & x \mid (e\ e) \mid \textbf{fun } x \rightarrow e \mid c \mid op \mid \textbf{let } x = e \textbf{ in } e \mid (e,e) \mid \ell \\ & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \textbf{if } e \textbf{ at } e \textbf{ then } e \textbf{ else } e \end{array}$$

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 lambda-abstraction that defines the function whose parameter is x and whose result is the value of e. Constants c are the integers, the booleans and we assume having a unique value: **()** that have the type *unit*. This is the result type of assignment (like in Objective Caml). The set of primitive operations *op* contains arithmetic operations, fix-point operator **fix**, test function **isnc** of **nc** (which plays the role of Objective Caml's `None`), our parallel operations (**mkpar**, **apply**, **put**) and our store operation **ref**, **!** and **:=**. We note e_1**:=**e_2 for **:=**(e_1, e_2). Locations are written ℓ, pairs (e, e). We also have two conditional constructs: usual conditional **if then else** and global conditional **if at then else**. We note $\mathcal{F}(e)$, the set of free variables of an expression e. **let** and **fun** are the binding operators and the free variables of a location is the empty set. It is defined by trivial structural induction on e.

Before presenting the dynamic semantics of the language, i.e., how the expressions of mini-BSML are computed to *values*, we present the values themselves. There is one semantics per value of p, the number of processes of the parallel machine. In the following, $\forall i$ means $\forall i \in \{0, \ldots, p-1\}$ and the expressions are *extended with enumerated parallel vectors:* $\langle e, \ldots, e \rangle$ (nesting of parallel vectors

is prohibited; our type system enforces this restriction [4]). The values of mini-BSML are defined by the following grammar:

v	::=	**fun** $x \rightarrow e$	functional value	$\mid$	c	constant	$\mid$	op primitive
	$\mid$	$\langle v, \ldots, v \rangle$	p-wide parallel vector value	$\mid$	(v, v)	pair value	$\mid$	ℓ location

3.2. Rules

The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we used a small-step semantics. It consists of a predicate between an expression and another expression defined by a set of axioms and rules called steps. The small-step semantics describes all the steps of the calculus from an expression to a value. We suppose that we evaluate only expressions that have been type-checked [4].

Unlike in a sequential computer and a sequential language, an unique store is not sufficient. We need to express the store of all our processors. We assume a finite set $\mathcal{N} = \{0, \ldots, p-1\}$ which represents the set of processors names and we write i for these names and $\bowtie$ for all the network. Now, we can formalize the location and the store for each processor and for the network. We write s_i for the store of processor i with $i \in \mathcal{N}$. We assume that each processor has a store and a infinite set of addresses which are different at each processor (we could distinguish them by the name of the processor). We write $S = [s_0, \ldots, s_{p-1}]$ for the sequence of all the stores of our parallel machine. The imperative version of the small-steps semantics has the following form: $e/S \rightharpoonup e'/S'$. We will also write $e/s \rightharpoonup e'/s'$ when only one store of the parallel machine can be modified.

We note $\stackrel{*}{\rightharpoonup}$, for the transitive closure of $\rightharpoonup$ and note $e_0/S_0 \stackrel{*}{\rightharpoonup} v/S$ for $e_0/S_0 \rightharpoonup e_1/S_1 \rightharpoonup e_2/S_2 \rightharpoonup \ldots \rightharpoonup v/S$. We begin the reduction with a set of empty stores $\{\emptyset_0, \ldots, \emptyset_{p-1}\}$ noted $\emptyset_\bowtie$. To define the relation $\rightharpoonup$, we begin with some axioms for two kinds of reductions:

1. $e/s_i \stackrel{i}{\rightharpoonup} e'/s'_i$ which could be read as "in the initial store s_i, at processor i, the expression e is reduced to e' in the store s'_i".

2. $e/S \stackrel{\bowtie}{\rightharpoonup} e'/S'$ which could be read as "in the initial network store S, the expression e is reduced to e' in the network store S'".

We write $s + \{\ell \mapsto v\}$ for the extension of s to the mapping of ℓ to v. If, before this operation, we have $\ell \in Dom(s)$, we can replace the range by the new value for the location ℓ. To define these relations, we begin with some axioms for the relation of head reduction. We write $e_1[x \leftarrow e_1]$ the expression obtained by substituting all the free occurrences of x in e_1 by e_2.

For a single process: $(\textbf{fun } x \rightarrow e)\ v \ /\ s_i \quad \stackrel{\varepsilon}{\underset{i}{\rightharpoonup}} \quad e[x \leftarrow v] \ /\ s_i \quad (\beta^i_{fun})$.

For the whole parallel machine: $(\textbf{fun } x \rightarrow e)\ v \ /\ S \quad \stackrel{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} \quad e[x \leftarrow v] \ /\ S \quad (\beta^{\bowtie}_{fun})$.

Rules (β^i_{let}) and $(\beta^{\bowtie}_{let})$ are the same but having **let** $x = v$ **in** e instead of **fun**. For primitive operators and constructs we have some axioms, the δ-rules. For each classical δ-rule, we have two new reduction rules: $e \ /\ s_i \stackrel{\varepsilon}{\underset{\delta_i}{\rightharpoonup}} e' \ /\ s'_i$ and $e \ /\ S \stackrel{\varepsilon}{\underset{\delta_\bowtie}{\rightharpoonup}} e' \ /\ S'$. Indeed, these reductions do not change the stores and do not depend on the stores (we omit these rules for lack of space and we refer to [3]). Naturally, for the parallel operators, we also have some δ-rules but we do not have those δ-rules on a single processor but only for the network (figure 1).

A problem appears with the **put** operator. The **put** operator is used for the exchange of values, in particular, locations. But a location could be seen as a pointer to the memory (a location is a memory's addresses). If we send a local allocation to a processor that does not have the location in its store, there is no reduction rule to apply and the program stops with an error (the famous *segmentation fault* of the C language) if it deferences this location (if it reads "out" of the memory). A dynamic solution is to communicate the value contained by the location and to create a new location for this value (as in the `Marshal` module of Objective Caml). This solution implies the renaming of locations that are communicated to other processes. For this, we define Loc the set of location of a value. It is defined by trivial structural induction on the value. We define how to add a sequence of pair of location and value to a store with:

$$s + \emptyset \ = \ s \text{ and } s + [\ell_0 \mapsto v_0, \ldots, \ell_n \mapsto v_n] \ = \ (s + \{\ell_0 \mapsto v_0\}) \ + [\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n].$$

We note $\varphi = \{\ell_0 \mapsto \ell'_0, \ldots \ell_n \mapsto \ell'_n\}$ for the substitution, i.e, a finite application from location ℓ_i to another location ℓ'_i with $\{\ell_0, \ldots \ell_n\}$ is the domain of φ.

Now we complete our semantics by giving the δ-rules of the operators on the stores and the ref-

$$(\mathbf{mkpar}\ v)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \langle (v\ 0), \ldots, (v\ (p-1))\rangle\ /\ S \qquad (\delta^{\bowtie}_{mkpar})$$

$$\mathbf{apply}(\langle v_0, \ldots, v_{p-1}\rangle, \langle v'_0, \ldots, v'_{p-1}\rangle)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \langle (v_0\ v'_0), \ldots, (v_{p-1}\ v'_{p-1})\rangle\ /\ S \qquad (\delta^{\bowtie}_{apply})$$

$$\mathbf{if}\ \langle \ldots, \overbrace{\mathbf{true}}^{n}, \ldots\rangle\ \mathbf{at}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} e_1\ /\ S \text{ if } v = n \qquad (\delta^{\bowtie}_{ifatT})$$

$$\mathbf{if}\ \langle \ldots, \overbrace{\mathbf{false}}^{n}, \ldots\rangle\ \mathbf{at}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} e_2\ /\ S \text{ if } v = n \qquad (\delta^{\bowtie}_{ifatF})$$

$$\mathbf{put}(\langle \mathbf{fun}\ dst \to e_0, \ldots, \mathbf{fun}\ dst \to e_{p-1}\rangle)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \langle r_0, \ldots, r_{p-1}\rangle\ /\ S' \qquad (\delta^{\bowtie}_{put})$$

where $S = [s_0, \ldots, s_{p-1}]$ and $S' = [s'_0, \ldots, s'_{p-1}]$ where $\forall j\ .\ s'_j = s_j + h'_0 + \ldots + h'_{p-1}$
where $h'_j = [\ell'_0 \mapsto v_0, \ldots, \ell'_n \mapsto v_n]$ and $h_j = \{(\ell_0, v_0), \ldots, (\ell_n, v_n)\}$
where $\ell_k \in \mathrm{Loc}(e_j)$ and $\{l_k \mapsto v_k\} \in s_j$ and $\varphi_j = \{\ell_0 \mapsto \ell'_0, \ldots, \ell_n \mapsto \ell'_n\}$ and $e'_j = \varphi_j(e_j)$ and
$\forall i\ r_i = (\mathbf{let}\ v^i_0 = e'_0[dst \leftarrow i]\ \mathbf{in} \ldots v^i_{p-1} = e'_{p-1}[dst \leftarrow i]\ \mathbf{in} f_i)$ where
$f_i = \mathbf{fun}\ x \to \mathbf{if}\ x = 0\ \mathbf{then}\ v^i_0\ \mathbf{else} \ldots \mathbf{if}\ x = (p-1)\ \mathbf{then}\ v^i_{p-1}\ \mathbf{else}\ \mathbf{nc()}$

Figure 1. Parallel δ-rules

erences. We need two kinds of reductions. First for a single processor, δ-rules are (δ^i_{ref}), $(\delta^i_{!})$ and $(\delta^i_{:=})$ (given in figure 2). Those operations work on the store of the processor where this operation is executed. The **ref** operation creates a new allocation in the store of the processor, the **!** operation gives the value contained in the location of the store and the **:=** operation changes this value by another one.

$$\mathbf{ref}(v)\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} \ell\ /\ s_i + \{\ell \mapsto v\} \quad \text{if } \ell \notin Dom(s_i) \qquad (\delta^i_{ref})$$

$$!(\ell)\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} s_i(\ell)\ /\ s_i \quad \text{if } \ell \in Dom(s_i) \qquad (\delta^i_{!})$$

$$\ell{:=}v\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} ()\ /\ s_i + \{\ell \mapsto v\} \quad \text{if } \ell \in Dom(s_i) \qquad (\delta^i_{:=})$$

$$\mathbf{ref}(v)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \ell^{\bowtie}\ /\ S' \quad \text{where } \begin{cases} S' = [s_0 + \{\ell^{\bowtie}_0 \mapsto \mathcal{P}_0(v)\}, \ldots, s_{p-1} + \{\ell^{\bowtie}_{p-1} \mapsto \mathcal{P}_{p-1}(v)\}] \\ \forall i.\ell^{\bowtie}_i \notin Dom(s_i) \end{cases} \qquad (\delta^{\bowtie}_{ref})$$

$$!(\ell^{\bowtie})\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} v\ /\ S \quad \text{where } \exists v\ \forall i\ s_i(\ell^{\bowtie}_i) = v \text{ and } \ell^{\bowtie}_i \in Dom(s_i) \text{ or } \exists v\ v = \mathcal{P}^{-1}(s_i(\ell^{\bowtie}_i)) \qquad (\delta^{\bowtie}_{!})$$

$$\ell^{\bowtie}{:=}v\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} ()\ /\ S' \quad \text{where } \begin{cases} S' = [s_0 + \{\ell^{\bowtie}_0 \mapsto \mathcal{P}_0(v)\}, \ldots, s_{p-1} + \{\ell^{\bowtie}_{p-1} \mapsto \mathcal{P}_{p-1}(v)\}] \\ \forall i.\ell^{\bowtie}_i \in Dom(s_i) \end{cases} \qquad (\delta^{\bowtie}_{:=})$$

$$\ell^{\bowtie}\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} \ell^{\bowtie}_i\ /\ s_i \qquad (\delta^{\bowtie}_{\mathrm{proj}})$$

Figure 2. "Store" δ-rules

For the whole network, we have to distinguish between the name of a location created outside a **mkpar** which is used in expressions and its "projections" in the stores of each process. We note $l^{\bowtie}$ in the first case and $l^{\bowtie}_i$ for its projection in the store of process i. When an expression outside a **mkpar** creates a new location, each process creates a new location (an address) on its store (rule $(\delta^{\bowtie}_{ref})$, figure 2) where $\mathcal{P}_i$ is a trivial function of projection for the parallel vectors of a value. With this function, we assure that there is no data from another processes in a store. The assignment of location $l^{\bowtie}$ (rule $(\delta^{\bowtie}_{:=})$, figure 2) modifies the values of locations $l^{\bowtie}_i$ using also the function of projection (which have no effect if the value do no contain any parallel vectors).

This rule is only valid outside a **mkpar**. But a reference created outside a **mkpar** can be affected and deferenced inside an **mkpar**. For assignment, the value can be different on each process. To allow this, we need to introduce a rule $(\delta^{\bowtie}_{\mathrm{proj}})$ (figure 2) which transforms (only inside an **mkpar** and at process i) the common name $l^{\bowtie}$ into its projection $l^{\bowtie}_i$. Notice that the affection or the deferencing of a location $l^{\bowtie}$ cannot be done inside a **mkpar** with rules $(\delta^i_{:=})$ and $(\delta^i_{!})$ since the condition $l \in Dom(s_i)$ does not hold. The use of the rule $(\delta^{\bowtie}_{\mathrm{proj}})$ is first needed.

The deferencing of $l^{\bowtie}$ outside a **mkpar** can only occur if the value held by its projections at each process is the same or if this value is the projection of a value which contain a parallel vector (a value

which cannot be modify by any process since nested of parallelism is forbidden [4]). This verification is done by rule ($\delta^{\bowtie}_{!}$) where $\mathcal{P}^{-1}$ is a trivial function of de-projection for the values stores one each processes which contains the projection values. This de-projection do no need any communication.

The complete definitions of our reductions are: $\xrightarrow{i} = \xrightarrow[i]{\varepsilon} \cup \xrightarrow[\delta_i]{\varepsilon}$ and $\xrightarrow{\bowtie} = \xrightarrow[\bowtie]{\varepsilon} \cup \xrightarrow[\delta_{\bowtie}]{\varepsilon}$. It is easy to see that we cannot always make a head reduction. We have to reduce in depth in the sub-expression. To define this deep reduction, we need to define two kinds of contexts. We refer to [3] for such definitions.

3.3. Cost model preserving semantics

In order to avoid the comparison of the values held by projections of a $l^{\bowtie}$ location in rule ($\delta^{\bowtie}_{!}$) we can forbid the assignment of such a location inside a **mkpar**. This can be done by suppressing rule ($\delta^{\bowtie}_{\text{proj}}$). But in this case, deferencing inside a **mkpar** is no longer allowed. Thus, we need to add a new rule: $\mathbf{!}(\ell^{\bowtie})\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} s_i(\ell^{\bowtie}_i)\ /\ s_i \quad \text{if } \ell^{\bowtie}_i \in Dom(s_i) \quad (\delta^{i\bowtie}_{!})$ and modify the rule ($\delta^{\bowtie}_{!}$) to suppress the comparison: $\mathbf{!}(\ell^{\bowtie})\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \mathcal{P}^{-1}(s_i(l^{\bowtie}_i))\ /\ S \quad \text{if } \ell^{\bowtie}_i \in Dom(s_i) \quad (\delta'^{\bowtie}_{!})$.

This rule is not deterministic but since assignment of a $l^{\bowtie}$ location is not allowed inside a **mkpar** the projections of a $l^{\bowtie}$ location always contain the same value. The cost model is now compositional since the new ($\delta'^{\bowtie}_{!}$) does not need communications and synchronization.

4. Conclusions and Future Work

The Bulk Synchronous Parallel ML allows direct mode Bulk Synchronous Parallel programming. The semantics of BSML were pure functional semantics. Nevertheless, the current implementation of BSML is the `BSMLlib` library for Objective Caml which offers imperative features. We presented in this paper semantics of the interaction of our bulk synchronous operations with imperative features. The safe communication of references has been investigated, and for this particular point, the presented semantics conforms to the implementation. To ensure safety, communications may be needed in case of assignment (but in this case the cost model is no longer compositional) or references may contain additional information used dynamically to ensure that dereferencing of references pointing to local values will give the same value on all processes. We are currently working on the typing of effects [10] to avoid this problem statically.

REFERENCES

[1] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[2] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 2003. to appear.

[3] F. Gava and F. Loulergue. Semantics of a Functional BSP Language with Imperative Features. Technical Report 2002-14, University of Paris Val-de-Marne, LACL, october 2002.

[4] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In *Parallel Computing Technologies (PaCT 2003)*, LNCS. Springer Verlag, 2003.

[5] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14*th IASTED PDCS Conference*, pages 452–457. ACTA Press, 2002.

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

[7] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[8] D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe and al., editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.

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

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

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