A Formal Semantics of the MULTI-ML Language

Conference Paper · June 2018

DOI: 10.1109/ISPDC2018.2018.00033

CITATION
1

READS
13

3 authors:

Victor Allombert
Université d'Orléans
9 PUBLICATIONS **30** CITATIONS
SEE PROFILE

Frédéric Gava
Université Paris-Est Créteil Val de Marne - Université Paris 12
64 PUBLICATIONS **403** CITATIONS
SEE PROFILE

Julien Tesson
Université Paris-Est Créteil Val de Marne - Université Paris 12
25 PUBLICATIONS **128** CITATIONS
SEE PROFILE

A formal semantics of the MULTI-ML language

Victor Allombert, Frédéric Gava, Julien Tesson

A formal semantics of the MULTI-ML language

Victor Allombert
Université d'Orléans, LIFO
Orléans, France
victor.allombert@univ-orleans.fr

Frédéric Gava
Université Paris-Est Créteil, LACL
Créteil, France
gava@u-pec.fr

Julien Tesson
Université Paris-Est Créteil, LACL
Créteil, France
julien.tesson@lacl.fr

***Abstract*—In the context of high performance computing, it is important to avoid indeterminism and dead-locks. MULTI-ML is a functional parallel programming language "à la ML", designed to program hierarchical architectures in a structured way. It is based of the MULTI-BSP bridging model. To ensure that a program "cannot go wrong", we first need to define how a program "goes". To do so, we propose a formal operational semantics of the MULTI-ML language to ensure the properties of the MULTI-BSP model. We first describe a core-language and then introduce the big step's semantics evaluation rules. Then, we propose a set of evaluation rules that describe the behaviour of the MULTI-ML language. The memory model is also precisely defined, as the MULTI-BSP model deals with multiple level of nested memories.**

Keywords : Semantics, MULTI-BSP, ML

I. Introduction

A. Context

Our previous work aimed at designing a parallel *functional* language based on the **B**ulk **S**ynchronous **P**arallelism (BSP) *bridging model* called BSML [1]. BSP is a model of parallelism which offers a high level of abstraction and takes into account real communications and synchronisation costs [2]. BSP has been used successfully for a broad variety of applications. To be compliant to a *bridging model* eases the way of writing codes that ensures *efficiency* and *portability* from one architecture to another. Thanks to a cost model it is also possible to reason on the algorithmic *costs*.

As modern HPC (**H**igh **P**erformance **C**omputing) architectures are *hierarchical* and have multiple layers of parallelism, communication between distant nodes cannot be as fast as among the cores of a given processor. As BSP was designed for *flat* architectures, we now consider the MULTI-BSP model [3], an extension of BSP which is dedicated to hierarchical architectures. MULTI-ML [4] is an extension of OCAML for programming MULTI-BSP algorithms. MULTI-ML uses a small set of primitives similarly to BSML for BSP algorithms.

To ensure that a language is properly implemented, it is important to provide a semantics that describes precisely the behaviour and the memory model of a language. Natural semantics (also called big-step semantics) aims to describe, in a formal way, how the overall result of the evaluation of an expression is obtained. In the context of MULTI-ML programming, it is thus possible to describe precisely the evaluations steps leading to a valid evaluation of a hierarchical computation. Using a set of inductive and co-inductive rules, we define both terminating and diverging terms evaluation. Thus, we are able to express formally how a MULTI-ML program "*goes*".

B. Outlines

In this article we introduce the formal (operational) semantics of the MULTI-ML language. First, we introduce MULTI-ML, the MULTI-BSP model and the previous models and languages, in section II. In (section III) we propose a core language that describes the syntax of the language, and the MULTI-ML execution model. Then, in section IV, we give and describe the terminating rules of the MULTI-ML semantics; followed be the diverging one. The formal properties of the semantics are discussed in section V. After the description of related works (in section VI) we conclude in section VII.

II. Previous work

A. The BSP bridging model

In the BSP model [5], a computer is a set of $\mathbf{p}$ *uniform* pairs of processor-memory with a communication network. A BSP program is executed as a *sequence* of *super-steps* (Fig. 1), each one divided into three successive disjointed phases: (1) each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; (2) the network delivers the requested data; (3) a *global synchronisation barrier* occurs, making the transferred data available for the next super-step. Thanks to this structured execution scheme, deadlocks are avoided and determinism is preserved.

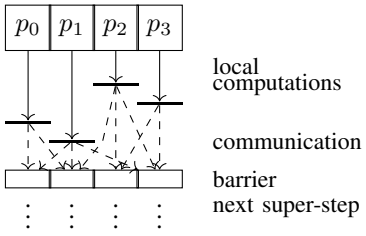

Fig. 1. A BSP super-step

As BSP architecture can be easily mapped on any general purpose parallel architecture, it is possible to accurately *estimate* the execution time of a BSP program with the BSP parameters (not detail in this article). The execution time (cost) of a super-step is the sum of the maximal of the local processing time, the data delivery and the global synchronisation times. The total cost of a BSP program is thus the sum of its super-steps's costs.

B. The BSML language.

BSML [1] is a functional parallel programming language designed to program BSP algorithms. It uses a *small set of primitives* and is currently implemented as a library for the ML (Meta Language) programming language OCAML.

A BSML program is built as a ML one but using a specific data structure called *parallel vector*. Its generic ML type is α `par`. A vector expresses that each of the $\mathbf{p}$ processors *embeds* a value of any type α. Informally, they work as follows: Let $\ll$ `e` $\gg$ be the vector holding `e` everywhere (on each processor), the $\ll$ $\gg$ indicates that we enter into the scope of a vector. The informal semantics of this construction is $\langle \mathtt{e}, \ldots, \mathtt{e} \rangle$, assuming that $\langle ... \rangle$ denotes a parallel vector.

Within a vector, the syntax `$x$` can be used to read the vector `x` and get the local value it contains. Informally, we have $\mathtt{v}_i$ on processor i, assuming that $\mathtt{v} \equiv \langle \mathtt{v}_0, \ldots, \mathtt{v}_{\mathbf{p}-1} \rangle$. The *ids* can be accessed with the predefined vector `pid`.

The **`proj`** primitive is the only way to *extract* local values from a vector. Given a vector, it returns a function such that applied to the *pid* of a processor, returns the value of the vector at this processor. The **`proj`** primitive is typed α `par`$\rightarrow$`(int`$\rightarrow\alpha$`)`. **`proj`** performs communication to make local results available globally and ends the current super-step. Informally, we have `proj` $\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto$ (`fun` $i \rightarrow x_i$).

The **`put`** primitive is another communication primitive. It allows any local value to be *transferred* to any other processor. It is also synchronous, and ends the current super-step. **`put`** is typed `(int`$\rightarrow\alpha$`)par`$\rightarrow$`(int`$\rightarrow\alpha$`)par`. The parameter of **`put`** is a vector that, at each processor, holds a function returning the data to be sent to processor j when applied to j. The result of **`put`** is another vector of functions: at a processor j the function, when applied to i, yields the value *received from* processor i by processor j. Informally, we have `put` $\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle$`fun` $i \rightarrow f_i$ $0, \ldots,$ `fun` $i \rightarrow f_i$ $(\mathbf{p}-1) \rangle$.

C. The Multi-BSP bridging model.

MULTI-BSP is a bridging *model* [3] which is adapted to hierarchical architectures, mainly *clusters* of *multi-cores*. It is an extension of the BSP bridging model. The structure and abstraction brought by MULTI-BSP allows to have portable programs with *scalable* performance predictions, without dealing with low-level details of the architectures. This model brings a *tree*-based view of nested components (*sub-machines*) of hierarchical architectures where the lowest stages (*leaves*) are processors and every other stage (*nodes*) contains memory.

Every component can execute code but they have to synchronise in favour of data exchange. Thus, MULTI-BSP does not allow subgroup synchronisation of any group of processors: at a stage i there is only a synchronisation of the sub-components, a synchronisation of each of the computational units that manage the stage $i-1$. So, a node executes some code on its nested components (*aka* "*children*"), then waits for results, does the communication and synchronises the sub-machine. A MULTI-BSP algorithm is thus composed by several super-steps, each step is synchronised for each sub-machine.

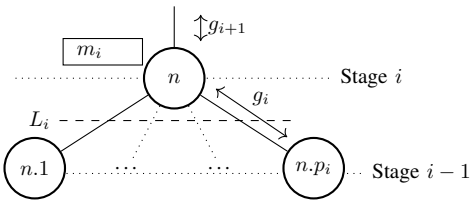

Fig. 2. The Multi-BSP parameters

Mainly, an instance of MULTI-BSP is defined by $\mathbf{d}$, the fixed depth of the (balanced and homogeneous) tree architecture, and by 4 parameters for each *stage* i of the tree : $(\mathbf{p_i}, \mathbf{g_i}, \mathbf{L_i}, \mathbf{m_i})$; described in Fig 2: $\mathbf{p_i}$ is the number of sub-components inside the $i-1$ stage; $\mathbf{g_i}$ is the *bandwidth* between stages i and $i-1$: the ratio of the number of operations to the number of words that can be transmitted in a second; $\mathbf{L_i}$ is the *synchronisation cost* of all sub-components of a component of $i-1$ stage; $\mathbf{m_i}$ is the amount of memory available at stage i for each component of this stage. Thanks to those parameters, the cost of a MULTI-BSP algorithm can be computed as the sum of the costs of the super-steps of the root node, where the cost of each of these super-steps is the maximal cost of the super-steps of the sub-components (plus communication and synchronisation); And so on.

D. The Multi-ML language.

MULTI-ML [6], [4] (https://git.lacl.fr/vallombert/Multi-ML) is based on the idea of executing BSML-like codes on every stage of a MULTI-BSP architecture. This approach facilitates *incremental* development from BSML codes to MULTI-ML ones. MULTI-ML follows the MULTI-BSP approach where the hierarchical architecture is composed by *nodes* and *leaves*. On nodes, it is possible to build parallel vectors, as in BSML. This parallel data structure aims to manage values that are stored on the sub-nodes: at stage i, the code **`let`** `v=`$\ll$ `e` $\gg$ evaluates the expression `e` on each $i-1$ stages. Inside a parallel vector, we note `#x#` to copy the value `x` stored at stage i to the memory $i-1$.

MULTI-ML also introduce the concept of *multi-function* to recursively go through a MULTI-BSP architecture. A *multi-function* is a particular recursive function, defined by the keyword **`let multi`**, which is composed by two codes: the node and the leaf codes. The *recursion* is initiated by calling the multi-function (recursively) inside the scope of a parallel vector, that is to say, on the sub-nodes. The evaluation of a multi-function starts (and ends) on the root node. The Fig. 3 shows how a multi-function is defined.

After the definition of the multi-function `mf` on line 1 where `[args]` symbolises a set of arguments, we define the node code (from line 2 to 6). The recursive call of the multi-function is done on line 5, within the scope of a parallel vector. The node code ends with a value `v`, which is available as a result of the recursive call from the upper node. The leaf code, from lines 7 to 9 consists of sequential computations.

```
let multi mf [args]=
| where node =
| | (* BSML code*)
| | ...
| | ≪ mf [args] ≫
| | ... in v
| where leaf =
| | (* OCaml code *)
| | ... in v
```

Fig. 3. A Multi-ML code.

e	::=		
		$\mathtt{x}$	*Variable*
	|	$\mathbf{op}$	*Operator*
	|	$\mathbf{cst}$	*Constant*
	|	$\mathbf{let}\ x = e\ \mathbf{in}\ e$	*Let binding*
	|	$\mathbf{fun}\ x \rightarrow e$	*Function*
	|	$\mathbf{rec}\ f\ x \rightarrow e$	*Recursive function*
	|	$\mathbf{multi}\ f\ x \rightarrow e \dagger e$	*Multi-function*
	|	$(e\ e)$	*Application*
	|	$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$	*Conditional*
	|	(e, e)	*Pairing*
	|	$\mathbf{mkpar}\ e$	*Parallel primitives*
	|	$\mathbf{proj}\ e \mid \mathbf{put}\ e$	*Synchro. parallel primitives*
	|	$\mathbf{replicate}\ (\mathbf{fun}\ \_ \rightarrow e\ )$	*Core parallel primitives*
	|	$\mathbf{down}\ x \mid \mathbf{apply}\ e\ e$	*Core parallel primitives*
v	::=		*Values*
	|	$\mathbf{op}$	*Operator*
	|	$\mathbf{cst}$	*Constant*
	|	$\overline{(\mathbf{fun}\ x \rightarrow e\ )[\mathcal{E}]}$	*Closure*
	|	$\overline{(\mathbf{rec}\ f\ x \rightarrow e\ )[\mathcal{E}]}$	*Recursive closure*
	|	$\overline{(\mathbf{multi}\ f\ x \rightarrow e \dagger e)[\mathcal{E}]}$	*Multi-function closure*
	|	(v, v)	*Pair*
	|	$< v, \ldots, v >$	*Parallel vector*
$\mathbf{op}$	::=		*Operators*
	|	$+, -, *, /, \mathbf{fst}, \mathbf{snd}, \ldots$	*Basic operators*
$\mathbf{cst}$	::=	$1, 2, ..., true, false, (), ...$	*Constants*
$\mathcal{E}$	::=	$\{x_1 \mapsto v_1, \ldots x_n \mapsto x_n\}$	*Environment*

Fig. 4. The μMULTI-ML grammar.

As expected, the synchronous communication primitives of BSML are also available to communicate values from/to parallel vectors. We also propose another parallel data structure called *tree*. A tree is a distributed structure where a value is stored in every nodes and leaves memories. A tree can be built using a multi-tree-function, with the **let multi tree** keyword and can be handled by several primitives of the language.

III. A CORE LANGUAGE

To ease the understanding and avoid to be overwhelmed by the details of a complete language, we choose to select a subset of MULTI-ML to form a core-language called μMULTI-ML. It relies on a minimal set of ML constructions and a set MULTI-ML primitives. This set is sufficient enough to express all the parallel behaviour that are used in MULTI-ML. Thus, features such as records, modules, pattern matching, sum types are excluded from μMULTI-ML. The grammar of the μMULTI-ML is defined in Figure 4.

In this grammar, $\mathtt{x}$ and f range over an infinite set of *identifiers*. We also find the typical ML-like language constructors such as **let** for bindings and also `fun` and `rec` for, respectively, function and recursive functions. As expected, the application is denoted $(e\ e)$. For the sake of readability, we take the liberty to use the familiar infix notation for binary and ternary operators, as well as the usual precedence and associativity rules. When the context is clear, we can avoid the usage of parentheses. The multi-function definition is written with the keyword **multi**. It takes one arguments and two expressions separated by the † symbol; the first one is used to control the parallelism on the node and the second is for leaf code. Then, we have the parallel primitive **mkpar** which aims to build parallel vectors; followed by the synchronous parallel primitives used for communications: **proj** and **put**.

The distinction made between the syntactic sugar (the $\ll$ $\gg$, $\#$ and $\$$ notations), used when programming MULTI-ML algorithms, and the core parallel primitives (**replicate**, **down** and **apply**), available in the semantics only, simplifies the semantics. Indeed, the syntactic sugar eases the way of programming but it is not suitable for the semantics as it introduces implicit assumptions. Thus, we must transform and abstract the syntactic sugar using the *core parallel primitives*. The transformation applied to switch from the syntactic sugar to the core parallel primitives is straightforward and produce and equivalent expression. The parallel vector scope, denoted $<< e >>$, is transformed using the **replicate** core primitive. Thus, $<< e >>$ is simply transformed into $\mathbf{replicate}\ (\mathbf{fun}\ \_ \rightarrow e\ )$. The $\$$ syntax is transformed using the **apply** primitive. Like in BSML, the transformation is simple and does not require a complicated expression analysis. To do so, we build a vector of functions that takes, as argument, the dollar's annotated value. Using the **apply** primitive, we can *apply* this vector of functions on the vector of values. For example, the expression $<< e\ \$x\$ >>$ is transformed into $\mathbf{apply}\ (\mathbf{replicate}\ (\mathbf{fun}\ \_\ x \rightarrow e\ x\ ))\ x$. The $\#$ syntax stands for a communication of a value to the sub-nodes. The primitive **down** is used to perform such a transfer. As the **down** primitive aims to distribute a value to all the sub-nodes, the transformation is obvious. For example, the expression $<< e\ \#x\# >>$ is transformed into $\mathbf{apply}\ (\mathbf{replicate}\ (\mathbf{fun}\ \_\ x \rightarrow e\ x\ ))(\mathbf{down}\ x)$. As the sharp annotated value is given as argument of the vector of functions, there are no redundant copies. The expression $<< \#x\# + \#x\# >>$ is transformed into a code that copy x to the sub-nodes, only once.

Then, the evaluation of an expression leads to a value $\mathtt{v}$. Variables stands for the standard operators, such as common computations on integers or the **fst** and **snd** projections, and also constants (which might refer to integer values, booleans, characters, strings, *etc.*), functional, recursive and multi-function closures (that is to say a value which stores both a function and its environment), pairs and also parallel vectors.

An environment $\mathcal{E}$ is interpreted as a partial mapping (informally called "finite mapping") with finite domain from identifiers to values. The extension of $\mathcal{E}$ by v in x, written $\mathcal{E} \oplus \{x \mapsto v\}$.

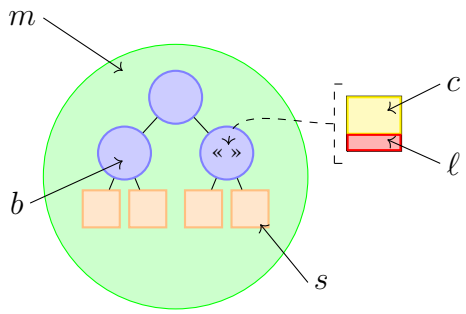

Fig. 5. The Multi-ML localities

A. The multi-level execution model

As MULTI-ML deals with MULTI-BSP architectures, we can distinguish several levels of execution. An expression is thus evaluated at a specific stage of the MULTI-BSP architecture, depending on the level of parallelism expressed. Those levels are useful to identify where the computations take place, but also where a data is stored. Furthermore, the notion of locality is embedded in the MULTI-ML type system to guarantee safe communications through the MULTI-BSP levels (the MULTI-ML type system is not discussed in this article). The memory localities (see Fig. 5) of MULTI-ML are the following:

- `m`: The *multi*, or global, memory of the multi-level architecture. At this level, all the nodes and leaves of the MULTI-BSP architecture are concerned by any execution. A value defined at level `m` is accessible "anywhere" on the machine.
- `b`: The BSP memory of a node. At this level, a BSML code can be executed in order to manage communications and parallel computations. It is possible to *point toward* the underlying memory via the parallel vectors. A value defined at level `b` is only accessible in the concerned node and can be communicated with the respect of certain limitations.
- `l`: The *local* memory of a sub-node (inside a parallel vector). This is the underlying memory that can be managed inside a parallel vector from the upper node . This memory can contain *pointers* to the upper memory via the `#` syntax. Expressions of this level cannot be communicated because they are specific to a node and may have no sense anywhere else.
- `c`: Like `l`, this memory is managed inside vectors but it is free of references to the upper level. It is possible to transfer such *communicable* values through the MULTI-BSP levels.
- `s`: The *sequential* memory of a leaf. It can contain specific values such as non-deterministic expressions. At this level, there is only sequential (ML) code and the execution takes place in the lower level of the memory. A value defined at level `s` can be communicated under certain constraints.

IV. BIG-STEP SEMANTICS

In this section we give the big-step semantics (also called natural semantics) to describe the evaluation of an expression to its final value using inductive rules. Here, we consider both a terminating semantics for terms where rules leads to a finite derivation tree; and diverging rules (leading to infinite evaluation). In the context of ML programming, we only consider an eager (or call-by-value) evaluation strategy.

We choose to describe a subset of the MULTI-ML semantics rules for conciseness reasons; all the rules can be found in [6].

A. Syntax definition

We denote the inference rules as following:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v} \text{ and } \frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

On the left-hand side, we have the inductive inference rule (leading to a finite derivation tree). It can be read: under the premises (or antecedents) $\mathcal{P}$, the inference rule concludes that, on the component p of locality $\mathcal{L}$, within a multi-environment $\mathcal{M}$, the expression e is evaluated into a value v. On the right-hand side, the co-inductive inference rule is written similarly, but using a double-bar. It stands for a case where e diverges: ∞. Otherwise, it is an error: no applicable rules are defined.

The environment of evaluation is denoted by $\mathcal{M}$ and is called a *multi-environment*. It is composed by two elements: (1) a MULTI-BSP environment and (2) a tree of environments. The first element represents the MULTI-BSP global *memory*. This memory contains values that are accessible by any processes. Thus every execution taking place outside the scope of a multi-function is global; otherwise, it is executed on nodes or leaves. Depending on the implementation and the architecture, it can be simulated by a virtual unified memory (each processor have a local copy) or by a physical global memory, for example, when simulating an execution in the toplevel. In practice, the memory is replicated on each component of the MULTI-BSP architecture. But for the semantics, we choose to create a single environment that is used during MULTI-BSP evaluations. An evaluation taking place at the MULTI-BSP level, denoted by $multi$, is then evaluated within the environment $||\mathcal{M}||@multi$. The notation $||\mathcal{M}||$ denotes the extraction of an environment from the memory $\mathcal{M}$ and the @ refers to the targeted memory. As expected, every unit of the architecture can access this MULTI-BSP environment. It is possible to lookup in it at any time, from any level p, that is to say, from each component of the MULTI-BSP architecture. Nevertheless, the modifications of this environment are possible at the MULTI-BSP level only.

The second element represents all the environments of the architecture. It takes the shape of a tree which maps all the environments of the different levels of the architecture, that is to say all the nodes and leaves. To refer to the environment of a particular component p of this tree is denoted $||\mathcal{M}||@p$. The components are identified using a regular tree numbering where 0 stands for the $root$ node, and 0.0 to $0.n$ identify its n sub-components. Thus, it contains its own environment only, deprived of the MULTI-BSP memory. As the MULTI-BSP memory is always available, the notation $||\mathcal{M}||_p$ stands for the environment composed by $||\mathcal{M}||@multi \cup ||\mathcal{M}||@p$, where the subscript p stands for the targeted memory.

$$\begin{array}{rcl} \mathcal{C}omm(\overline{(\mathbf{fun}\ x \to e\ )[\mathcal{E}]}) & = & \mathcal{C}omm\_expr_{\mathcal{E}}(e) \\ \mathcal{C}omm(\overline{(\mathbf{rec}\ f\ x \to e\ )[\mathcal{E}]}) & = & \mathcal{C}omm\_expr_{\mathcal{E}}(e) \\ \\ \mathcal{C}omm\_expr_{\mathcal{E}}(x) & = & \top, \mathbf{if}\ x \notin \mathcal{E} \\ & & \mathcal{C}omm(v), \mathbf{if}\ \{x \mapsto v\} \in \mathcal{E} \end{array}$$

Fig. 6. The communication predicates.

Now, we introduce the symbol of rule evaluation. The piece of notation $\Downarrow_p^{\mathcal{L}}$ denotes an evaluation taking place at a specific level $\mathcal{L}$, also parametrised by the position p. The evaluation level $\mathcal{L}$ can take the values `m`, `b`, `l` or `s`, to specify on which level the evaluation must occur. `m` is used when evaluation level is MULTI-BSP; `b` when it a BSP evaluation; `l` when the evaluation takes place within the scope of a parallel vector; and `s` for sequential evaluations.

In the semantics, we do not make any distinctions between `l` and `c`. It is not useful, as we propose a predicate to check if a value is communicable: $\mathcal{C}omm(v)$ (in Fig. 6), for rules performing communications. By abuse of the notation, this predicate is $\top$ if the value v is communicable, $\bot$ otherwise.

When communicating closures, we check if the expression e contained in the closure is communicable within the enclosed environment. To do so, we use the $\mathcal{C}omm\_expr$ predicate which is pretty straightforward: given an environment $\mathcal{E}$, on **`mkpar`**, **`proj`**, **`put`**, **`apply`**, **`down`**, **`replicate`** and multi-functions it is $\bot$; otherwise it calls, recursively, the predicate on sub-expression.

The subtlety of this notation comes with the communicability of x. We assume that, if $x \notin \mathcal{E}$, then the value x is communicable. Indeed, if x does not belong to $\mathcal{E}$, it means that x is a free variable which comes from the application of a closure, as the environment used for a closure is ε, which is the environment of the closed term. Thus, during the evaluation of an application $(e_1\ e_2)$, we assume that x is communicable in e_1 as its communicability will be checked with e_2. As expected, the communication of $\mathbf{cst}$ and $\mathbf{op}$ is $\top$ and on multi-function closures it is $\bot$.

The position p stands for the identifier of the component where an evaluation takes place. We use the notation $p.i$ to refer to the ith sub-unit of a component that is accessible through a parallel vector. The identifier alias $root$ stands for the root node and, as explained previously, $multi$ is used when the whole architecture is concerned by an evaluation.

B. Terminating evaluation rules

As the MULTI-ML language proposes different levels associated with different behaviours, we propose several rules that are either generic or specific to `m`, `b`, `l`.

1) Generic rules: The generic rules that can be evaluated regardless to the evaluation level can be found in Fig. 7.

The VALUES and OP_EVAL rules a straightforward.

The VAR rule introduces the predicate $lookup(x, \mathcal{M}, p, \mathcal{L})$ (defined in Fig. 8). As the VAR rule gives the value corresponding to a binding, we must *lookup* for it in a particular way. The value x must be seek, first, in $||\mathcal{M}||@p$, which is the memory of the evaluation at level p. Then,

$$\text{VALUES}\quad \frac{}{\mathcal{M} \vdash \mathbf{cst} \Downarrow_p^{\mathcal{L}} \mathbf{cst}}$$

$$\text{VAR}\quad \frac{\{x \mapsto v\} \in lookup(x, \mathcal{M}, p, \mathcal{L})}{\mathcal{M} \vdash x \Downarrow_p^{\mathcal{L}} v}$$

$$\text{CLS}\quad \frac{\begin{array}{l}\mathcal{E} = select(\mathcal{M}, \mathcal{F}(\mathbf{fun}\ x \to v\ ), p, \mathcal{L}) \\ v \equiv \overline{(\mathbf{fun}\ x \to e\ )[\mathcal{E}]}\end{array}}{\mathcal{M} \vdash \mathbf{fun}\ x \to e\ \Downarrow_p^{\mathcal{L}} v}$$

$$\text{APP}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\mathbf{fun}\ x \to e\ )[\mathcal{E}]} \quad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \\ \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \vdash e \Downarrow_p^{\mathcal{L}} v'\end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathcal{L}} v'}$$

$$\text{LET\_IN}\quad \frac{\begin{array}{l}\mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} v_1 \\ \mathcal{M} \oplus_p \{x \mapsto v_1\} \vdash e_2 \Downarrow_p^{\mathcal{L}} v_2\end{array}}{\mathcal{M} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow_p^{\mathcal{L}} v_2}$$

Fig. 7. Generic evaluation rules.

$$\begin{array}{l} lookup(x, \mathcal{M}, p, \mathcal{L}) = \\ \quad \mathbf{if}\ \{x \mapsto v\} \in ||\mathcal{M}||@p\ \mathbf{then}\ v \qquad \textit{Value at } p \\ \quad \mathbf{if}\ \{x \mapsto v\} \in ||\mathcal{M}||@multi\ \mathbf{then} \\ \quad\ \mathbf{if}\ \mathcal{L} \equiv \mathtt{l}\ \mathbf{then}\ \bot \qquad \textit{Out of reach} \\ \quad\ \mathbf{else}\ v \qquad \textit{Value at } multi \\ \quad \mathbf{else}\ \bot \qquad \textit{Unbound value} \end{array}$$

Fig. 8. The lookup predicate.

if the value is not found there, we must *look* for it in the MULTI-BSP environment: $||\mathcal{M}||_{multi}$. If the value is not found anyway, the identifier is unbound in the current expression, which is an impossible (or $\bot$) evaluation (*"unbound value"*). As expected, if the evaluation takes place at the MULTI-BSP level ($multi$), the variable x is only searched for in the environment $||\mathcal{M}||_{multi}$. Furthermore, we add the locality of the evaluation as an input of the $lookup$ predicate to ensure that the values defined in the `m` memory will not be accessed within the scope of a parallel vector (of locality `l`).

The CLS (or CLOSURE) rule is used to enclose all the needed variables that are available in $\mathcal{M}$. To do so, we use the predicate $select(\mathcal{M}, \mathcal{V}, p, \mathcal{L})$ (Fig. 9) to *select* all the variables of the set of variables $\mathcal{V}$ from the multi-environment $\mathcal{M}$. As the predicate uses $lookup$, we must provide the current level p and its locality $\mathcal{L}$. The set of variables $\mathcal{V}$ is determined using $\mathcal{F}(e)$, which stands for the set of free variables of the expression. The set of free variables $\mathcal{F}(e)$ is defined as usual, and available in Fig. 10.

The APP rule is used to evaluate functional constructions on values. The rule introduces the notation $\oplus_p$, which is a

$$\begin{array}{l} select(\mathcal{M}, \mathcal{V}, p, \mathcal{L}) = \\ \quad let\ \mathcal{E}' = \emptyset \\ \quad \forall \alpha \in \mathcal{V}, lookup(\alpha, \mathcal{M}, p, \mathcal{L}) \oplus \mathcal{E}' \\ \quad where\ \mathcal{V} \equiv \alpha_0, \ldots, \alpha_n \\ \quad return\ \mathcal{E}' \end{array}$$

Fig. 9. The select predicate.

$\mathcal{F}(e)$ ::=

$$
\begin{array}{rcl}
\mathcal{F}(\mathbf{op}) & = & \emptyset \\
\mathcal{F}(\mathbf{cst}) & = & \emptyset \\
\mathcal{F}(x) & = & x \\
\mathcal{F}(\mathbf{fun}\ x \rightarrow e\ ) & = & \mathcal{F}(e) \backslash \{x\} \\
\mathcal{F}(\mathbf{rec}\ f\ x \rightarrow e\ ) & = & \mathcal{F}(e) \backslash \{x\} \\
\mathcal{F}(\mathbf{multi}\ f\ x \rightarrow e_1 \dagger e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \backslash \{x\} \\
\mathcal{F}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \backslash \{x\} \\
\mathcal{F}(e_1\ e_2) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
\mathcal{F}((e_1, e_2)) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2)
\end{array}
$$

Fig. 10. The free variable predicate.

$$\text{REPLICATE}\quad \frac{\forall i \in p \quad \mathcal{M} \oplus_{p_i} \{f \mapsto (\overline{\mathbf{fun}\ \_ \rightarrow e\ [\,]})\} \vdash f\ ()\ \Downarrow^{\mathtt{l}}_{p_i} v_i \quad \mathcal{C}omm(\overline{(\mathbf{fun}\ \_ \rightarrow e\ [\,])})}{\mathcal{M} \vdash \mathbf{replicate}\ (\mathbf{fun}\ \_ \rightarrow e\ ) \Downarrow^{\mathtt{b}}_{p} < v_0, \ldots, v_n >}$$

$$\text{DOWN}\quad \frac{\mathcal{M} \vdash x \Downarrow^{\mathtt{b}}_{p} v \quad \mathcal{C}omm(v)}{\mathcal{M} \vdash \mathbf{down}\ x \Downarrow^{\mathtt{b}}_{p} < v >}$$

$$\text{MKPAR}\quad \frac{\mathcal{M} \vdash e \Downarrow^{\mathtt{b}}_{p} \overline{(e')[\mathcal{E}']} \quad \forall i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ i \Downarrow^{\mathtt{b}}_{p} v_i \quad \mathcal{C}omm(v_i)}{\mathcal{M} \vdash \mathbf{mkpar}\ e \Downarrow^{\mathtt{b}}_{p} < v_0, \ldots, v_n >}$$

$$\text{APPLY}\quad \frac{\mathcal{M} \vdash e_1 \Downarrow^{\mathtt{b}}_{p} < \overline{(e_i)[\mathcal{E}_i]} > \qquad \mathcal{M} \vdash e_2 \Downarrow^{\mathtt{b}}_{p} < v_0, \ldots, v_n > \quad \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}, x_i \mapsto v_i\} \vdash f_i\ x_i \Downarrow^{\mathtt{l}}_{p.i} v'_i}{\mathcal{M} \vdash \mathbf{apply}\ e_1\ e_2 \Downarrow^{\mathtt{b}}_{p} < v'_0, \ldots, v'_n >}$$

$$\text{PROJ}\quad \frac{\mathcal{M} \vdash e \Downarrow^{\mathtt{b}}_{p} < v_0, \ldots, v_n > \quad \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f \mapsto \overline{(e')[\mathcal{E}']}\} \vdash f\ i \Downarrow^{\mathtt{b}}_{p.i} v_i \quad \mathcal{C}omm(v_i)}{\mathcal{M} \vdash \mathbf{proj}\ e \Downarrow^{\mathtt{b}}_{p} \overline{(e')[\mathcal{E}']}}$$

$$\text{PUT}\quad \frac{\mathcal{M} \vdash e \Downarrow^{\mathtt{b}}_{p} < \overline{(e_0)[\mathcal{E}_0]}, \ldots, \overline{(e_n)[\mathcal{E}_n]} > \quad \forall i, j \in p \quad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ j \Downarrow^{\mathtt{l}}_{p.i} v_{ij} \quad \mathcal{M} \oplus_{p.j} \{f'_j \mapsto \overline{(e'_j)[\mathcal{E}'_j]}\} \vdash f'_j\ i \Downarrow^{\mathtt{l}}_{p.j} v_{ij}}{\mathcal{M} \vdash \mathbf{put}\ e \Downarrow^{\mathtt{b}}_{p} < \overline{(e'_0)[\mathcal{E}'_0]}, \ldots, \overline{(e'_n)[\mathcal{E}'_n]} > f}$$

Fig. 11. BSP evaluation rules.

concatenation operator allowing to enrich the environment of p with bindings. Here, we add the closure environment $\mathcal{E}$ and the binding $\{x \mapsto v\}$ to $||\mathcal{M}||_p$ to the environment of p.

The LET_IN rule is necessary to bind values within a context. The rule is straightforward as it only adds $\{x \mapsto v_1\}$ to the environment of p.

2) BSP rules: In Fig.11 we describe the BSP rules that can be executed at level `b` (that is to say on nodes).

The REPLICATE rule is used to create parallel vectors. This primitive is introduced by the code transformation and is not available in the syntax of the MULTI-ML language. The REPLICATE primitive requires, syntactically, a function definition as an argument. This function takes *any* argument, denoted with _ (similarly to the OCAML syntax), which is unbound in the enclosed term. Such function definition allows to delay the evaluation of the expression e when REPLICATE is evaluated. As the expression e is generated (by the transformation from syntactic sugar to core primitives), there are no free variables in e. Indeed, a variable appears within the scope of a parallel vector, it was necessarily annotated by `#` or `$`, and thus bound by the **`apply`** primitive. Otherwise, the expression is ill-formed. This is necessary for the MULTI-ML runtime system. The notation $\forall i \in p$ allows to reference the i sub-components of the current node p, from 0 to n. Thus, the closure is added to the environment of each leaf of the current node, identified by p_i, if the term is communicable. Then, the function is evaluated on the unit value $()$, on each sub-components. As the evaluation takes place within the scope of a parallel vector, its evaluation locality is `l`. The resulting value is, as expected, a parallel vector which contains the results of the evaluation of e on each sub-component.

The DOWN rule is used to distribute data to the sub-nodes. From a variable x, it builds a parallel vector containing the corresponding value v, if v is communicable. The distributed values are identical on the sub-nodes, thus the resulting parallel vector is $< v >$.

MKPAR also aims to build a parallel vector. Nevertheless, the evaluation of the ith elements of the resulting vector takes place at level p, sequentially. This evaluation is done within a single environment that is common to every evaluation. Thus, one evaluation could impact the environment of another one with a code using side effects, even if there is no way of doing any side effects in the current semantics. It is important to note that the big-step semantics does not allow to describe precisely the evaluation order of the i evaluations. We arbitrarily choose to do it in accordance to the ascending order: from 0 to n. Thus, the expression e' is applied on i, which corresponds to the processor identifier (*pid*) of the i^{st} component. It is important to check that every value v_i is communicable. Indeed, we need to communicate the values $v_0, .., v_n$ to the sub-nodes of the level p and we must check the validity of such a transfer.

As expected, the APPLY rule applies a parallel vector of functions (e_i) on a parallel vector of data (v_i) and returns a parallel vector of values. Here, we compute, in parallel, the evaluations of $f_i\ v_i$ within the environment of each sub-component i.

The PROJ rule must be applied on a parallel vector available at the current level p. The resulting value is a closure which contains the content of the given vector. As expected, the values are accessible via their original identifiers. It is also necessary to check whether or not the values of the given vector are communicable in order to build a communicable closure.

The PUT rule works as expected. It takes a parallel vector of communication functions, referenced as f_i within environment p_i. Then, the received values are accessible via f'_j. Here, both i and j stand for the sub-components of the current node p. They are denoted as distinct variables to show the *all-to-all* exchange.

3) Local rules: Rules that can be executed within the scope of a parallel vector (of locality `l`) can be found in Fig. 13.

The MULTI_NODE and MULTI_LEAF rules are relative to the recursive multi-function calls, which corresponds to the

$$\text{MULTI\_NODE}\ \frac{\begin{array}{l} isNode(p) \quad \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']} \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \oplus_p \\ \quad \{f \mapsto \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_p^{\mathtt{b}} v' \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathtt{l}} v'}$$

$$\text{MULTI\_LEAF}\ \frac{\begin{array}{l} isLeaf(p) \quad \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']} \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \vdash e_2' \Downarrow_p^{\mathtt{s}} v' \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathtt{l}} v'}$$

Fig. 12. Local evaluation rules.

$$\text{MULTI\_DEF}\ \frac{\begin{array}{l} \mathcal{M}' = select(||\mathcal{M}||_{multi}, \mathcal{F}(\textbf{multi}\ f\ x \to e_1 \dagger e_2)) \\ v \equiv \overline{(\textbf{multi}\ f\ x \to e_1 \dagger e_2)[\mathcal{M}']} \end{array}}{\mathcal{M} \vdash (\textbf{multi}\ f\ x \to e_1 \dagger e_2) \Downarrow_{multi}^{\mathtt{m}} v}$$

$$\text{MULTI\_CALL}\ \frac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_{multi}^{\mathtt{m}} \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']} \\ \mathcal{M} \vdash e_2 \Downarrow_{multi}^{\mathtt{m}} v \\ \mathcal{M} \oplus_{root} \overline{\{x \mapsto v\}} \oplus_{root} \\ \quad \{f \mapsto \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_{root}^{\mathtt{b}} v' \\ \mathcal{C}omm(v') \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_{multi}^{\mathtt{m}} v'}$$

Fig. 13. MULTI-BSP evaluation rules.

application of multi-function function on a value within the scope of a parallel vector. MULTI_NODE and MULTI_LEAF are applied regarding to the type of the current component. The operators $isNode(p)$ and $isLeaf(p)$ are used to make such a distinction. Informally speaking, $isNode(p)$ is impossible (returns $\perp$) if p has no sub-components. On the contrary, $isLeaf(p)$ is impossible ($\perp$) if p has sub-components. Concerning the MULTI_NODE rule, the environment of evaluation of e_1' (standing for the node code) is enriched by the argument given to the multi-function and the closure of the multi-function. Indeed, the sub-node must be able to recall the multi-function. We can observe that, after enriching the environment, the evaluation of e_1' goes from `l` to `b` (from a vector to a (sub-)node). This locality change is due to the fact that, when a recursive call of a multi-function (done within the scope of a parallel vector) is evaluated, we must initiate the recursion on the sub-nodes: which corresponds to evaluate e_1' at level `b`. A level change occurs during the evaluation of this rules. On the contrary, the MULTI_LEAF rule is not concerned by this environment enrichment because of its status of leaf. A similar change of level can be observed in the MULTI_LEAF rule, when we go from `l` to `s` (from a vector to a (sub-)leaf).

4) Multi-BSP rules: Finally, the rule used to call a multi-function is defined in Fig. 13.

The MULTI_DEF rule aims to create the closure of a multi-function and make it available in the MULTI-BSP environment. To do so, we simply call the predicates $select(||\mathcal{M}||_{multi}, \mathcal{F}(\textbf{multi}\ f\ x \to e_1 \dagger e_2))$ in order to built the environment $\mathcal{M}'$ from the MULTI-BSP environment and the free variables of the multi-function itself.

The MULTI_CALL rule is used to initiate the multi-function evaluation. As expected, the multi-function evaluation starts on the *root* node, with an environment enriched by the given argument and the multi-function closure. Thus, the evaluation of e_1', corresponding to the node code, is evaluated at level `b`, on the $root$ node. As the resulting value of the evaluation of the multi-function is going to be transferred to the MULTI-BSP level, we also need to check if v' is communicable. Indeed, the final value must be available in the $||\mathcal{M}_{multi}||$ environment. To do so, in case of a distributed MULTI-BSP memory, we must *broadcast* v' from the $root$ node to all the components of the MULTI-BSP architecture. Thus, we must check that v' is communicable.

$$\text{MKPAR-E}\ \frac{\mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} \infty}{\mathcal{M} \vdash \texttt{mkpar}\ e \Downarrow_p^{\mathtt{b}} \infty}$$

$$\text{MKPAR-V}\ \frac{\mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} \overline{(e')[\mathcal{E}']} \quad \exists i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ i \Downarrow_p^{\mathtt{b}} \infty}{\mathcal{M} \vdash \texttt{mkpar}\ e \Downarrow_p^{\mathtt{b}} \infty}$$

Fig. 14. **mkpar** coinductive evaluation rule.

C. Diverging evaluation rules

Using the co-inductive approach described in [7], we propose a set of rules which describe the diverging rules. The diverging rules are pretty straightforward as they describe the fact that each evaluation of an expression of a rule can lead to infinite evaluation. For conciseness reasons, we only describe (in Fig.14) the **mkpar** rule (all the diverging rules can be found in [6]). The MKPAR-E rules describes the divergence of the evaluation of e; whereas MKPAR-V describes the divergence of e', which is the expression enclosed in the closure given by the evaluation of e. In both cases, the evaluation of the whole rule diverges.

V. SEMANTICS PROPERTIES

In this section, we give the properties of the semantics rules. We prove that the evaluation of an expression following the proposed rules is deterministic. The lemmas 1 and 2 ensure that, if a value v is obtained from the execution of e and then, the evaluation is deterministic. The lemma 3 ensures that an evaluation produces a value or diverges (evaluation and divergence exclusivity), that is to say, and evaluation "*does not go wrong*".

Lemma 1 (Evaluation is deterministic)**.** *Let e be a program, let $\mathcal{M}$ be an environment, $\mathcal{L}$ an evaluation locality, p a position and v_1 and v_2 be values. If $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_1$ and $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2$ then $v_1 = v_2$.*

Proof. See [6] □

Lemma 2 (Evaluation or not)**.** *Let e be a* MULTI-ML *expression, $\mathcal{M}$ be an evaluation environment, $\mathcal{L}$ an evaluation locality and p a position. Then:*

- *It is impossible to obtain a value v such that $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$*
- **or** *there exists a value v such that $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$*

Proof. (**Using classical logic**). The excluded middle holds over $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$. □

Lemma 3 (Evaluation does not go wrong)**.** *Let e be a* MULTI-ML *program, let $\mathcal{M}$ be an environment, $\mathcal{L}$ an evaluation locality, p a position and v a value. If $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$ and $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty$ then there is a contradiction.*

Proof. See [6] □

Note that those lemmas does not prove that any program can be evaluated. For example, the program $0\ 0$ (zero applied on zero) goes wrong since, for any environment $\mathcal{E}$ and value v, neither $\mathcal{E} \vdash (0\ 0) \Downarrow v$ nor $\mathcal{E} \vdash (0\ 0) \Downarrow_\infty$ holds. To do that, we must provide a typing system which allow to identify such programs.

VI. Related Work

A. Hierarchical programming and Multi-BSP libraries

There are many papers about the gains of mixing shared and distributed memories —*e.g.* MPI and OPEN-MP [8]. As intended, the programmer must manage the distribution of the data for these two different models. For example, the work of [9] in which a BSP extension of C++ runs the same code on both a cluster and on multi-cores. But it is the responsibility of the programmer to avoid harmful nested parallelism. This is thus not a dedicated language working for hierarchical architectures. We can also highlight the work of NESTSTEP [10] which is a C/JAVA library for BSP computing, which authorises nested computations in case of a cluster of multi-cores — but without any safety.

B. Distributed functional languages.

Except in [11], there is a lack of comparisons between parallel functional languages. It is difficult to compare them since many parameters have to be taken into account: efficiency, scalability, expressiveness, *etc.* A data-parallel extension of HASKELL called NEPAL has been done in [12], an abstract machine is responsible for the distribution of the data over the available processors. MULTIMLTON [13] is a multi-core aware runtime for standard ML, which is an extension of the MLTON compiler. It manages composable and asynchronous events using, in particular, *safe-futures*. A description of other bridging models for hierarchical architectures and other parallel languages can be found in [6].

Currently, we are not aware of any safe and efficient functional parallel language dedicated to hierarchical architectures.

VII. Conclusion

In this paper we have presented the formal semantics of the MULTI-ML language, a parallel functional language dedicated to hierarchical architectures, relying of the MULTI-BSP bridging model. The semantics describes the behaviour of the MULTI-ML language regarding the multiple level of execution of hierarchical architectures. Furthermore, we propose a memory model that ensures consistency with respect to the MULTI-BSP programming model. Using both a terminating semantics leading to a finite derivation tree and a diverging semantics leading to infinite evaluations, we have described how a MULTI-ML expression "*goes*". Moreover, in accordance to the presented lemmas, we ensure that the evaluation of a valid MULTI-ML expression "*does not go wrong*".

References

[1] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski, "Bulk synchronous parallel ML with exceptions," *Future Generation Computer Systems*, vol. 26, no. 3, pp. 486–490, Mar. 2010.

[2] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, "Questions and Answers about BSP," *Scientific Programming*, vol. 6, no. 3, pp. 249–274, 1997.

[3] L. G. Valiant, "A Bridging Model for Multi-core Computing," *J. Comput. Syst. Sci.*, vol. 77, no. 1, pp. 154–166, Jan. 2011.

[4] V. Allombert, F. Gava, and J. Tesson, "Multi-ML: Programming Multi-BSP Algorithms in ML," *International Journal of Parallel Programming*, p. 20, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01160164

[5] L. G. Valiant, "A Bridging Model for Parallel Computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[6] V. Allombert, "Functional Abstraction for Programming Multi-Level Architectures: Formalisation and Implementation," Ph.D. dissertation, Université Paris Est, Créteil, France, Jul. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01693568

[7] X. Leroy and H. Grall, "Coinductive big-step operational semantics," *Information and Computation*, vol. 207, no. 2, pp. 284–304, Feb. 2009.

[8] F. Cappello and D. Etiemble, "MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=370049.370071

[9] K. Hamidouche, J. Falcou, and D. Etiemble, "A Framework for an Automatic Hybrid MPI+OpenMP Code Generation," in *Proceedings of the 19th High Performance Computing Symposia*. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 48–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=2048577.2048584

[10] C. W. Kessler, "NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model," *The Journal of Supercomputing*, vol. 17, no. 3, pp. 245–262, Nov. 2000.

[11] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á. J. Rebón, and P. W. Trinder, "Comparing Parallel Functional Languages: Programming and Performance," *Higher Order Symbolic Computing*, vol. 16, no. 3, pp. 203–251, Sep. 2003.

[12] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel, "Nepal - Nested Data Parallelism in Haskell," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. London, UK, UK: Springer-Verlag, 2001, pp. 524–534. [Online]. Available: http://dl.acm.org/citation.cfm?id=646666.699740

[13] K. C. Sivaramakrishnan, L. Ziarek, and S. Jagannathan, "MultiMLton: A multicore-aware runtime for standard ML," *Journal of Functional Programming*, vol. 24, no. 06, pp. 613–674, 2014. [Online]. Available: http://journals.cambridge.org/article_S0956796814000161