A Parallel Virtual Machine for Bulk Synchronous Parallel ML

Frédéric Gava and Frédéric Loulergue

Laboratory of Algorithms, Complexity and Logic – University Paris Val-de-Marne
61, avenue du général de Gaulle – 94010 Créteil cedex – France
{gava,loulergue}@univ-paris12.fr

Abstract. We have designed a functional data-parallel language called BSML for programming bulk-synchronous parallel (BSP) algorithms. The execution time can be estimated and dead-locks and indeterminism are avoided. The BSMLlib library has been implemented for the Objective Caml language. But there is currently no full implementation of such a language and an abstract machine is needed to validate such an implementation. Our approach is based on a bytecode compilation to a parallel abstract machine performing exchange of data and synchronous requests derived from the ZAM, the efficient abstract machine of the Objective Caml language.

1 Introduction

Bulk Synchronous Parallel ML or BSML is an extension of ML 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 [18] 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* when its physical process structure is made explicit. Such algorithms offer predictable and scalable performances and BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [11]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

Our `BSMLlib` library implements the BSML primitives using Objective Caml [9] and MPI [17]. It is efficient [10] and its performance follows curves predicted by the BSP cost model (the cost model estimates parallel execution times).

This library is used as the basis for the Caraml project, which aims to use Objective Caml for Grid computing with, for example, applications to parallel databases and molecular simulation. In such a context, security is an important issue, but in order to obtain security, safety must be first achieved. An abstract machine is used for the implementation of Caml and is particular easy to prove correct w.r.t. the dynamic semantics [5]. 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. A polymorphic type system and a type inference has been designed and proved correct w.r.t. a small-steps semantics.

A parallel abstract machine [13] for the execution of BSML programs has been designed and proved correct w.r.t. the BSλ-calculus [11], using an intermediate semantics. Another abstract machine [12] has been designed but those machines are not adapted for grid computing and security because the compilation schemes need the static number of processes (this is not possible for Grid computing) and some instructions are not realistic for real code and a real implementation. The novelty of this paper is the presentation of an abstract machine without this drawbacks. This machine is an extension of the Zinc Abstract Machine [7] (ZAM) which is the virtual machine used in the implementations of the Objective Caml [9] and Caml-light languages and which is very efficient.

We first present the BSP model and give an informal presentation of BSML through the `BSMLlib` programming library (section 2). Then we present the ZAM (section 3), we extend it to a bulk synchronous parallel abstract machine and we define the compilation of BSML to this machine (section 4).

2 Functional Bulk Synchronous Parallelism

Bulk Synchronous Parallelism The Bulk Synchronous Parallel (BSP) model [18] describes: an abstract parallel computer, a model of execution and a cost model.

A BSP computer has three components: 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. The performance of the BSP computer is characterized by three parameters (often 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 one word at most).

A BSP program is executed as a sequence of *super-steps*, each one divided into three successive and logically disjoint phases (at most): (a) each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes, (b) the network delivers the requested data transfers, (c) a global synchronization barrier occurs, making the transferred data available for the next super-step.

This structured parallelism allows accurate performance prediction through a cost model, which motivates the BSP model. Nevertheless we will not present it here for the sake of conciseness, but we refer to [16].

The `BSMLlib` library 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. A type system enforces this restriction [4]. The BSML parallel constructs operates on parallel vectors which 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`.

The distinction between a communication request and its realization at the barrier is ignored. `put` expresses communication and synchronization phases:

```
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)`. Global conditional is necessary of express algorithms like:

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

This framework is a good tradeoff for parallel programming because: we defined a *confluent calculus* so we designed a purely functional parallel language from it. Without side-effects, programs are easier to prove, and to re-use. An eager language allows good performances ; this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

3 The Zinc Abstract Machine

Abstract machines for the λ-calculus To calculate the values of the λ-calculus, a lot of abstract machines have been designed. The first was the **SECD** machine [6] which was used for the first implementation of the LISP language. It uses **environment** (a list of values) for the closure and four stacks for the calculus. But it is an old and not optimized machine. In the same spirit, [2] presented the functional abstract machine (**FAM**). The **FAM** optimizes access to the environment by using arrays (constant cost access). The **G-machine** was designed for functional languages with a call by name strategy [15]. But we have an eager language so those techniques are not suitable for us. An interesting machines is the **CAM**, categorical abstract machine, which was introduced and used by Curien to implement the **CAML** language [3] (a variant of Standard ML [14]). A extension of this machine for BSP computing has been done by [12].

[7] introduced a powerful abstract machine, the **ZAM** (Zinc abstract machine, Zinc = Zinc Is Not CAM) which underlies the **bytecode interpreter** of Objective Caml (and Caml-light). This machine was derived from the Krivine's abstract machine and from the λ-calculus with **explicit substitution** [5]. This machine is interesting because its instructions could be "easily" translated to efficient bytecode (with some optimizations like "threaded code") and also to native code [8].

In the terminology of Peyton-Jones [15], the **ZAM** is an environment and closure based abstract machine following the *push-enter* model (unlike the **CAM** and the **SECD** which used a *eval-apply* models) and a *call-by-value* evaluation strategy i.e. arguments are evaluated and pushed before the function and then according to the number of arguments, return a closure or the evaluation of the code of the function with its arguments. This method optimizes allocation of the arguments and the evaluation of our expressions.

The ZAM The machine state has four components: (1) a code pointer c representing the code being executed as a sequence of instruction; (2) an environment e: a sequence of machine value $v_1 \dots v_n$ associating the value v_i to the variable having de Bruijn indices i (they transform an identifier to the number of λ-abstraction which are included between the identifier and the λ-abstraction that binds it; this method was used to solve the problem of binding variables); (3)

a stack s (a sequence of machine values and return contexts) holding function arguments, intermediate results, and function return contexts; (4) an integer n counting the number of function arguments available on the stack.

The manipulated machine-level values v are pointer to heap blocks written [T:$v_1 \ldots v_n$] where T is a tag attached to the block (an integer) and $v_1 \ldots v_n$ are the values contained in the block. We use tags $1 \ldots n$ to encode the constructor C of inductive types (supposed to be declared) and a distinct tag T_λ for function closures. The values of the BSML language are represented by: (a) a primitive constant or (b) a heap blocks, a function value, by a closure [T_λ:c,e] of the compiled code c for the function body, and an environment e associating to the variables free in the body or (c) an inductive type $\mathbf{C}(\overrightarrow{v})$, by the heap block [#C,($\overrightarrow{v}$)] where $\overrightarrow{v}$ are the representations of the values of the tuple $\overrightarrow{v}$ and #C is the tag number associated with the constructor C.
The transition of the abstract machine and the halting configurations are show in Figure 1 (two lines for before and after the execution of the instruction).

4 The BSP-Zinc Abstract Machine

Abstract machines for the BSλ-calculus For BSλ-calculus, [13] modified the **SECD**. But this new machine still has the same problems as the original one: slowness, difficulty to have real instruction machine and optimize it, notably for the exchange of closures. To remedy to these problems, [12] introduced a modification of the **CAM** for BSλ-calculus. But this machine has two problems: the number of processors of the machine which will execute the program has to be known at the compilation phase (it is not at all adapted for eases portability, in *Grid* computing for example) and the instruction for the exchange of values is difficult to translate to real code because this instruction adds instructions to the code during execution. The first problem is specific to [12] but the second problem is shared with the BSP **SECD** machine. We give here an abstract machine which is an extension of the **ZAM**, suitable for BSP computing and which has not those drawbacks. Furthermore, this abstract machine will be the basis of a complete implementation of the BSML language.

The BSP ZAM The BSP ZAM has two kinds of instructions: sequential and parallel ones. Its corresponds to the two structures of the original calculus: BSλ-calculus [11]. The BSP ZAM is obtained by duplicating the sequential **ZAM** on each process. This allows the machine to execute asynchronous computations phases of BSP super-steps. To express the other phases of BSP super-steps we need another set of instructions: synchronous ones. For the first phase of the BSP model (asynchronous computations), we also need an instruction for the number of processes: **Nprocs** and in the spirit of SMPD programming, the names of each process: **Pid**. The last instruction is needed for the construction of the parallel vectors by giving the name of the process (i). Then, to express the synchronization and communication phases of the BSP super-step, we need to add two special instructions to the BSP ZAM: **At** and **Send**. They are the only

Code	Environment	Stack	Number of arguments
Access(i);c	e	s	n
c	e	e(i).s	n
Quote(i);c	e	s	n
c	e	i.s	n
Closure(c');c	e	s	n
c	e	[T_λ:c',e].s	n
Push(c');c	e	s	n
c	e	$\langle$ c',e,n $\rangle$.s	n
Apply(i)	e	[T:c',e'].s	n
c'	e'	s	i
Grab;c	e	v.s	n+1
c	v.e	s	n
c_0=Grab;c	e	$\langle$c',e',n'$\rangle$.s	0
c'	e'	[T_λ:c_0,e].s	n'
Return	e	v.$\langle$c',e',n' $\rangle$.s	0
c'	e'	v.s	n'
Return	e	[T:c',e'].s	n if $n > 0$
c'	e'	s	n
Cons(m);c	e	$v_1 \ldots . v_m$.s	n
c	e	$(v_1, \ldots, v_m)$.s	n
Makeblock(T,m);c	e	$v_1 \ldots . v_m$.s	n
c	e	[T:$v_1 \ldots v_m$].s	n
Switch($c_1, \ldots, c_m$)	e	[T:$v_1, \ldots, v_p$].s	n if $1 \leq T \leq m$
c_T	$v_p \ldots . v_1$.e	s	0
Branch(c_1,c_2)	e	true.s	n
c_1	e	s	n
Branch(c_1,c_2)	e	false.s	n
c_2	e	s	n
CloseR(c');c	e	s	n
c	e	v.s	n where v=[T_λ:c',v.e]
Add;c	e	$n_1.n_2$.s	n
c	e	n.s	n where $n = n_1 + n_2$
Equal;c	e	$v_1.v_2$.s	n where v=true
c	e	v.s	n if $v_1 = v_2$ false else
Proj(i,j);c	e	$(v_1, \ldots, v_i, \ldots, v_j)$.s	e
c	e	v_i.s	e

and the halting configuration:

Code	environment	stack	number of arguments	result value
Return	e	v	0	v
c_0=Grab;c	e	ε	0	[$T_\lambda : c_0$,e]

Fig. 1. Sequential ZAM instructions

instructions which need BSP synchronization between the sequential ZAM on each process. **At** (with a **Branch** instruction) is used for the global conditional. **Send** is an instruction for the primitive synchronous **put** operator, used for the exchange of values between the processes and here the **ZAM** machines. The instructions of the BSP **ZAM** are given in Figure 2 for a p processors machine and only for the codes and the stacks (environment and number of arguments do not change).

Codes	Stacks
$\langle$ **At**;c_0, ..., **At**;c_{p-1} $\rangle$ $\langle$ c_0, ..., c_{p-1} $\rangle$	$\langle$ $n.t_0.s_0$, ..., $n.t_{p-1}.s_{p-1}$ $\rangle$ $\langle$ $t_n.s_0$, ...,$t_n.s_{p-1}$ $\rangle$
$\langle$ ..., **Nprocs**;c, ... $\rangle$ $\langle$ ..., c, ... $\rangle$	$\langle$ ..., s, ... $\rangle$ $\langle$ ..., $p.s$, ... $\rangle$
$\langle$ ..., $\overbrace{\textbf{Pid};c}^{i}$, ... $\rangle$ $\langle$ ..., $\overbrace{c}^{i}$, ... $\rangle$	$\langle$ ..., $\overbrace{s}^{i}$, ... $\rangle$ $\langle$ ..., $\overbrace{i.s}^{i}$, ... $\rangle$
$\langle$ **Send**;c_0, ..., **Send**;c_{p-1} $\rangle$ $\langle$ c_0, ..., c_{p-1} $\rangle$	$\langle$ $t_0^0 :: t_0^1 :: \ldots :: t_0^{p-1}.s_0$, ..., $t_{p-1}^0 :: \ldots :: t_{p-1}^{p-1}.s_{p-1}$ $\rangle$ $\langle$ $t_0^0 :: t_1^0 :: \ldots :: t_{p-1}^0.s_0$, ..., $t_0^{p-1} :: \ldots :: t_{p-1}^{p-1}\rangle.s_{p-1}$ $\rangle$

and where i is the name of the process and p the number of processes.

Fig. 2. BSP ZAM instructions

Compilation of BSML In order to be concrete, we shall consider the problem of compiling our core language to the machine. The BSML language uses real identifiers and here we used De Bruijn indices codings for these. An index of the variable is thus needed. The compilation scheme for the **ZAM** is presented as a function $[\![e]\!]_c$, where e is an expression and c an instruction sequence representing the continuation of e. It returns an instruction sequence that evaluates e, leaves its value at the top of the stack, and continues in sequence by executing the code c. We suppose that the expressions are well-typed and the nested of parallel vectors is rejected by the type checker [4] of the BSML language.

Sequential mini-BSML expressions A variable x is compiled to an **Access** instruction carrying the de Bruijn index of the variable. The execution of **Access** looks up the i^{th} entry in the machine environment and pushes it on the stack. In the same spirit, constants are trivially compiled to a **Quote** and **Nprocs** instruction. For the primitive operators we use an extra-function *Inst* which gives the instruction of each operator; for example: $\mathit{Inst}(\mathbf{op}) = \mathbf{Add}$ or **Equal** where $\mathbf{op} = +$ or $=$.

A curried function compiles to a **Closure** instruction, which at run-time builds a closure of its arguments with the current environment, and pushes the closure on the stack. The arguments of **Closure** is the code for the body e of the function, preceded by m **Grab** instructions and followed by a **Return** instruction. The code for a multiple application $e\ e_1\ \ldots e_m$ first pushes a return

frame containing the code to be executed when the applied function returns, as well as the current environment and argument count (instruction **Push**). Then, the arguments and the function are evaluated right-to-left, and their values pushed on the stack. Finally the **Apply** instruction branches to the code of the closure obtained by evaluating e setting the argument count to m.

The conditional is compiled with a **Branch** instruction, according to whether the top of the stack is **true** or **false**, executed a code. For inductive types, the compilation and execution of constructor applications is straightforward: the arguments of the constructor are evaluated, and the **Makeblock** instruction creates the representation of the constructed terms, tagged with the constructor number. For the *case* statement, a return frame to the continuation c is pushed first. The **Switch** instruction then discriminates on the tag of the matched value, and branches to the code of corresponding cases arm, after adding the fields of the matched value to the environment, thus binding the pattern variables. The **Return** instruction at the end of the code for each arm then restores the original environment and branches back to the continuation c. For recursive function, we used a **CloseR** instruction to constructs a recursive closure. In the simplified presentation given in this paper, this is a cyclic closure $v = [T_\lambda : c, v.e]$ where the first slot of the environment, corresponding to the recursive variable f in the source term, point back to the closure itself (usual implementation uses the scheme described in [1] instead of cyclic closures). Now we give the compilation of the sequential **ZAM** (Figure 3).

$$
\begin{array}{ll}
[\![x]\!]c & = \mathbf{Access}(i);c \quad \text{where i is the De Burijn index} \\
[\![\mathbf{const}]\!]c & = \mathbf{Quote}(\mathbf{const});c \\
[\![\mathbf{nproc}]\!]c & = \mathbf{Nprocs};c \\
[\![\mathbf{op}]\!]c & = \mathbf{Closure}(\mathbf{Grab};\mathbf{Access}(1);\mathit{Inst}(\mathbf{op});\mathbf{Return};);c \\
[\![\mathbf{fun}\ x_1 \ldots x_m \rightarrow e]\!]c & = \mathbf{Closure}(\overbrace{\mathbf{Grab};\ldots\mathbf{Grab};}^{\text{m times}}[\![e]\!];\mathbf{Return});c \\
[\![e\ e_1 \ldots e_m]\!]c & = \mathbf{Push}(c);[\![e_m]\!]\ldots[\![e_1]\!];[\![e]\!];\mathbf{Apply}(m); \\
[\![C(e_1,\ldots,e_m)]\!]c & = [\![e_m]\!];\ldots[\![e_1]\!];\mathbf{Makeblock}(n,\#\mathbf{C});c \\
[\![(e_1,\ldots,e_n)]\!]c & = [\![e_n]\!];\ldots[\![e_1]\!];\mathbf{Cons}(n);c \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]c & = \mathbf{Push}(c);[\![e_1]\!];\mathbf{Closure}(\mathbf{Grab};[\![e_2]\!];\mathbf{Return});\mathbf{Apply}(1); \\
[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]c & = \mathbf{Push}(c);[\![e_1]\!];\mathbf{Branch}([\![e_2]\!];\mathbf{Return};,[\![e_3]\!];\mathbf{Return}); \\
[\![\mathbf{let\ rec}\ f\ x_1 \ldots\ x_n = e_1\ \mathbf{in}\ e_2]\!]c & = \mathbf{Push}(c);\mathbf{CloseR}(\overbrace{\mathbf{Grab};\ldots;\mathbf{Grab};}^{\text{n times}}[\![e_1]\!];\mathbf{Retun}); \\
 & \quad\ \mathbf{Closure}(\mathbf{Grab};[\![e_2]\!];\mathbf{Return});\mathbf{Apply}(1); \\
[\![\mathbf{case}\ e\ \mathbf{of}\ (\mathbf{C}_i(\overrightarrow{x_1}) \rightarrow b_1,\ldots)]\!]c & = \mathbf{Push}(c);[\![e]\!];\mathbf{Switch}([\![b_1]\!];\mathbf{Return};,\ldots);
\end{array}
$$

Fig. 3. Compilation of the sequential **ZAM**

Parallel operators For the primitive operations, we used what the semantics suggests: the parallel operator **mkpar** is compiled to the application of the

expression to the "pid" (or name) of the processes and **apply** is simply the application because the first value is supposed to be a closure (or recursive) from an abstraction or an operator:

$$[\![\mathbf{mkpar}]\!] = \mathbf{Closure}(\mathbf{Grab};\mathbf{Push}(\mathbf{Return};);\mathbf{Pid};\mathbf{Access}(1);\mathbf{Apply}(1););c$$
$$[\![\mathbf{apply}]\!]c = \mathbf{Closure}(\mathbf{Grab};\mathbf{Grab};\mathbf{Push}(\mathbf{Return};);\ \mathbf{Access}(1);\mathbf{Access}(2);\mathbf{Apply}(1););c$$

The global conditional is compiled like the traditional conditional but with another argument and by adding the **At** instructions before the **Branch** to make the synchronous and communication running of the BSP model:

$$[\![\mathbf{if}\ e_1\ \mathbf{at}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4]\!]c = \mathbf{Push}(c);[\![e_1]\!];[\![e_2]\!];\mathbf{at};\ \mathbf{Branch}([\![e_3]\!];\mathbf{Return};,[\![e_4]\!];\mathbf{Return};);$$

The compilation of the **put** operator is the only real difficulty. To compile the **put** operator, a first way presented by [12] used a compiling scheme with a static number of processes and two special instructions was added: one adds codes at the running time to calculate all the values to send and a second exchanges those values and generated a code to read them. Clearly, in a real implementation with real machine codes this is not easy to generate a lot of machine codes essentially when the number of processes is large. To remedy to this problem, we can remark that to calculate the values to send and read them, we always do the same things. The trick is to generate **ZAM** codes that calculate and read the values by iteration. To do this, we can add a special closure name *put_function* to iterate the calculus. We can write it in our functional language in an extended syntax to directly have the code generated by our compiler:

```
fun f->let rec create n = if n=0 then [f n] else (f n)::(create (n-1))
  in let construct = (fun g -> fun i -> fun value -> fun n ->
      if n=i then value else (g n)) in
let rec read liste_v n = match liste_v with
   []     -> fun x -> None             (* case not process !!! *)
 | hd::tl -> if n=0 then (fun pid -> if pid=0 then hd else None)
             else (construct (read tl (n-1)) n hd)
in read (create (bsp_p()-1)) (bsp_p()-1));;
```

`create` recursively computes the value to send. `read` and `construct` build recursively the code of the return of the **put** operator. To completely compile the `put` primitive operator, the compiling function has to add manually the **Send** instructions in the code generated from the *put_function* between the end of the construction and the call of the read function: $[\![\mathbf{put}]\!]_c = \text{Insere\_Send}([\![\mathit{put\_function}]\!]_c)$

5 Conclusions and Future Work

The Bulk Synchronous Parallel ZINC Abstract Machine presented here provides a detailed and portable model of parallel environments management for Bulk Synchronous Parallel ML. It has two advantages with respect to the BSP-SECD machine and the BSP-CAM of [12] : the number of processes of the parallel machine has not to been known at compilation, thus improving the portability ; the

communication operation does not add instructions at execution, making the implementation both simpler and more classic. The next phases of the project will be: (a) the proof of correctness of this machine with respect to BSML semantics, (b) the parallel implementation of this abstract machine. This BSP ZAM implementation will be the basis of a parallel programming environment developed from the Caml-light language and environment. It will include our type inference [4] and will thus provide a very safe parallel programming environment.

Acknowledgments. This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML (www.caraml.org). The authors wish to thank the anonymous referees for their comments.

References

1. A.W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.
2. L. Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 208–217, Austin, Texas, August 1984. ACM.
3. G. Cousineau and G. Huet. The caml primer. Technical Report 122, INRIA, 1990.
4. F. Gava and F. Loulergue. Synthèse de types pour Bulk Synchronous Parallel ML. In *Journées Francophones des Langages Applicatifs (JFLA 2003)*, january 2003.
5. T. Hardin, L. Maranget, and L. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.
6. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 4(6):308–320, 1964.
7. X. Leroy. The ZINC experiment: An economical implementation of the ML language. Rapport Technique 117, 1991.
8. X. Leroy. The caml special light system: modules and efficient compilation for caml. Technical Report 2721, INRIA, Novembre 1995.
9. Xavier Leroy. The Objective Caml System 3.06, 2002. web pages at www.ocaml.org.
10. F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14^{th} *IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.
11. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
12. A. Merlin and G. Hains. La Machine Abstraite Catégorique BSP. In *Journées Francophones des Langages Applicatifs*. INRIA, 2002.
13. A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, august 2001.
14. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
15. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
16. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.
17. M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
18. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.