A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML

Gava, F. and Louergue, F.

Technical Report TR-2002-20

Laboratory of Algorithms, Complexity and Logic
University of Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
F-94010 CRÉTEIL Cedex – FRANCE
Tel: +33 (0)1 45 17 16 47
Fax: +33 (0)1 45 17 66 01
A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML

Frédéric Gava and Frédéric Loulergue
Laboratory of Algorithms, Complexity and Logic
University Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
94010 Créteil cedex – France
Tel: +33 (0)1 45 17 16 50
Fax: +33 (0)1 45 17 66 01
{gava, loulergue}@univ-paris12.fr

December 12, 2002

Abstract

We have designed a functional data-parallel language called BSML for programming bulk-synchronous parallel (BSP) algorithms in so-called direct mode. In a direct-mode BSP algorithm, the physical structure of processes is made explicit. The execution time can then 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 natural semantics and a bytecode compilation to a parallel abstract machine performing exchange of data and synchronous requests derived from the abstract machine of the Caml language.

Keywords: Parallel Programming, Bulk Synchronous Parallelism, Functional Programming, Abstract Machine, Compilation

1 Introduction and related works

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 [31] 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 [15] 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 BSL calculus [22]: 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 [20] and MPI [30]. It is efficient [21] 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 was used for the implementation of Caml and is particular easy to prove correct w.r.t. the dynamic semantics [17] and with the compile scheme [3]. 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 [26] for the execution of BSML programs has been designed and proved correct w.r.t. the BSλ-calculus [22], using an intermediate semantics. Another abstract machine [25] 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.

We first present the BSP model and give an informal presentation of BSML through the BSMLlib programming library (section 2). We then present a formal definition of BSML and its dynamics semantics (section 3). Then we define a bulk synchronous parallel categorical abstract machine and the compilation of BSML to this parallel abstract machine (section 4) and conclude (section 5).

2 Functional Synchronous Parallelism

2.1 Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model [31, 24, 29] 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). The network can deliver an $h$-relation (communication phase where every processor receives/sends $h$ words at most ) in time $g \times h$.

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. The execution time of a super-step $s$ is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\text{Time}(s) = \max_{i: \text{processor}} w_{i}^{(s)} + \max_{i: \text{processor}} h_{i}^{(s)} \times g + l$$

where $w_{i}^{(s)}$ = local processing time on processor $i$ during super-step $s$ and $h_{i}^{(s)} = \max\{h_{i-1}^{(s)}, h_{i}^{(s)}\}$ where $h_{i}^{(s)}$ (resp. $h_{i}^{(s)}$) is the number of words transmitted (resp. received) by processor $i$ during super-step $s$.

The execution time of a program is the sum of the execution time of each of its super-steps. The execution time $\sum_{s} \text{Time}(s)$ of a BSP program composed of $S$ super-steps is therefore a sum of 3 terms:

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

where $W = \sum_{s} \max_{i} w_{i}^{(s)}$ and $H = \sum_{s} \max_{i} h_{i}^{(s)}$. In general $W$, $H$ and $S$ are functions of $p$ and of the size of data $n$, or of more complex parameters like data skew. To minimize execution time the BSP algorithm design must jointly minimize the number $S$ of super-steps, the total volume $h$, imbalance of communication, the total volume $W$ and imbalance of local computation.

2.2 Bulk Synchronous Parallel ML

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 underlying architecture. In particular, it offers the function \texttt{bsp.p:unit\rightarrow int} such that the value of \texttt{bsp.p()} is \texttt{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 \texttt{'}a par\texttt{'} which represents the type of \texttt{p}-wide parallel vectors of objects of type \texttt{'}a\texttt{',} one per process. The nesting of \texttt{par} types is prohibited. Our type system enforces this restriction [14]. This improves on the earlier design DPML/Caml Flight [16, 11] in which the global parallel control structure \texttt{sync} had to be prevented \texttt{dynamically} from nesting.

The BSML parallel constructs operate on parallel vectors. Those parallel vectors are created by

\[
\texttt{mkpar: \ (int \ \rightarrow \ 'a) \ \rightarrow \ 'a par}
\]

so that \texttt{(mkpar \ f)} stores \texttt{(f \ i)} on process \texttt{i} for \texttt{i} between 0 and \texttt{(p - 1)}. We usually write \texttt{f} as \texttt{fun \ pid\rightarrow e} to show that the expression \texttt{e} may be different on each processor. This expression \texttt{e} is said to be \texttt{local}. The expression \texttt{(mkpar \ f)} is a parallel object and it is said to be \texttt{global}.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a superstep) and phases of global communication (second phase of a superstep) with global synchronization (third phase of a superstep). Asynchronous phases are programmed with \texttt{mkpar} and with

\[
\texttt{apply: \ ('}a \ \rightarrow \ 'b\texttt{) \ \rightarrow \ 'a par \ \rightarrow \ 'b par}
\]

\texttt{apply (mkpar \ f) (mkpar \ e)} stores \texttt{(f \ i) \ (e \ i)} on process \texttt{i}. Neither the implementation of BSMLlib, nor its semantics prescribe a synchronization barrier between two successive uses of \texttt{apply}. Readers familiar with BSPlib will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by

\[
\texttt{put: (int\rightarrow 'a \ \rightarrow \ par \ \rightarrow \ (int\rightarrow 'a \ \rightarrow \ par}
\]

where \texttt{'}a option\texttt{'} is defined by: \texttt{type \ 'a \ option=\texttt{None} \ | \ Some \ of \ 'a.} \texttt{

Consider the expression: \texttt{put(mkpar(fun \ i\rightarrow fs,))}

\[
(*)
\]

To send a value \texttt{v} from process \texttt{j} to process \texttt{i}, the function \texttt{fsj} at process \texttt{j} must be such as \texttt{(fsj \ i)} evaluates to \texttt{Some \ v}. To send no value from process \texttt{j} to process \texttt{i}, \texttt{(fsj \ i)} must evaluate to \texttt{None}. Expression \texttt{(*)} evaluates to a parallel vector containing a function \texttt{fdi} of delivered messages on every process. At process \texttt{i}, \texttt{(fdi \ j)} evaluates to \texttt{None} if process \texttt{j} sent no message to process \texttt{i} or evaluates to \texttt{Some \ v} if process \texttt{j} sent the value \texttt{v} to the process \texttt{i}. The full language would also contain a synchronous conditional operation:

\[
\texttt{ifat: \ (bool \ par) \ \rightarrow \ int \ \rightarrow \ 'a \ \rightarrow \ 'a
\]

such that \texttt{ifat (v,i,v1,v2)} will evaluate to \texttt{v1} or \texttt{v2} depending on the value of \texttt{v} at process \texttt{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 BSMLib contains the function: \texttt{at: bool \ par \ \rightarrow \ int \ \rightarrow \ bool} to be used only in the construction: \texttt{if (at \ vec \ pid) \ then... \ else...} where \texttt{(vec:bool \ par)} and \texttt{(pid:int)}. If \texttt{at} expresses communication and synchronization phases. Global conditional is necessary of express algorithms like:

\[
\text{Repeat}
\]

\[
\text{Parallel Iteration}
\]

\[
\text{Until Max of local errors < \epsilon}
\]

Without it, the global control cannot take into account data computed locally.

3 Syntax and evaluation of mini-BSML

For the sake of conciseness, we limit our study to a subpart of the BSML language. It is an attempt to trade between integrating the principal features of functional and BSP language. This section introduces a core language, its syntax and its dynamic semantics.
3.1 Syntax

The expressions of mini-BSML (like in [6]), written e possibly with a subscript, have the following abstract syntax:

\[
\begin{align*}
  e &::= x & \text{variable} & \quad \text{fun} \; x \rightarrow e & \quad \text{function abstraction} \\
  \text{c} & \quad \text{constant} & \quad \text{let} \; x = e \; \text{in} \; e & \quad \text{local binding} \\
  \text{op} & \quad \text{primitive operation} & \quad \text{let rec} \; f \; x = e \; \text{in} \; e & \quad \text{recursive local binding} \\
  (e \; e) & \quad \text{application} & \quad \text{if} \; e \; \text{then} \; e \; \text{else} \; e & \quad \text{conditional} \\
  (e, \; e) & \quad \text{pair} & \quad \text{if} \; e \; \text{at} \; e \; \text{then} \; e \; \text{else} \; e & \quad \text{global conditional}
\end{align*}
\]

In this grammar, x ranges over a countable set of identifiers. Constants c are the integers, the booleans or the special constants of number of processes. The set of primitive operations op contains arithmetic operations, pair projection, test function isnc of the nc constant (which plays the role of Objective Caml’s None) and our parallel operations (mkpar, apply, put).

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. In the following, \( \forall i \) means \( \forall i \in \{0, \ldots, p-1\} \). There is one semantics per value of \( p \), the number of processes of the parallel machine. An environment \( \mathcal{E} \) is an application from variables to values. The values of mini-BSML are defined by the following grammar:

\[
\begin{align*}
  v &::= [x, \text{e}, \mathcal{E}] & \quad \text{closure value} & \quad \text{c} & \quad \text{constant} \\
  &\ | \quad [x, \text{e}, \mathcal{E}]_{\text{rec}} & \quad \text{recursive closure value} & \quad \text{op} & \quad \text{primitive} \\
  &\ | \quad (v, \text{v}) & \quad \text{pair value} & \quad (v, \ldots, v) & \quad \text{p-wide parallel vector value}
\end{align*}
\]

Functional values are represented by the well-known closure which contains an environment, the body and the binding variable of the function. The environment in a closure gives the value of the free variables of the body.

3.2 Dynamic semantics

The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we use the formalism of relational semantics (or natural semantics [18]). It consists of a predicate between expressions, an environment and values defined by a set of axioms and inference rules called evaluation judgments. An evaluation judgment tells whether an expression evaluates to a given result. There are two kinds of inductive rules, the first for the abstract syntax and the second for primitive operators. As the implementation of the BSMLlib, we do not use some constructors of the language for the parallel operations but primitive operators. Now we give the first inductive rules:

\[
\begin{align*}
  \mathcal{E} &\vdash \text{e} \Rightarrow \mathcal{E}(\text{e}) \\
  \mathcal{E} &\vdash \text{c} \Rightarrow \text{c} \\
  \mathcal{E} &\vdash \text{op} \Rightarrow \text{op} \\
  \mathcal{E} &\vdash \text{fun} \; x \rightarrow e \Rightarrow [x, \text{e}, \mathcal{E}] \\
  \mathcal{E} &\vdash \text{e}_1 \Rightarrow [x, \text{e}, \mathcal{E}'] \\
  \mathcal{E} &\vdash \text{e}_2 \Rightarrow v \\
  \mathcal{E} &\vdash \{x \mapsto v\} \Rightarrow v' \\
  \mathcal{E} &\vdash (\text{e}_1, \text{e}_2) \Rightarrow v'
\end{align*}
\]

\[
\begin{align*}
  \mathcal{E} &\vdash \text{e}_1 \Rightarrow [f, \text{fun} \; x \rightarrow e, \mathcal{E}']_{\text{rec}} \\
  \mathcal{E} &\vdash \text{e}_2 \Rightarrow v \\
  \mathcal{E} &\vdash \{f \mapsto [f, \text{fun} \; x \rightarrow e, \mathcal{E}]\}_{\text{rec}} \\
  \mathcal{E} &\vdash \{x \mapsto v\} \Rightarrow v' \\
  \mathcal{E} &\vdash \text{let} \; x = \text{e}_1 \; \text{in} \; \text{e}_2 \Rightarrow v
\end{align*}
\]

\[
\begin{align*}
  \mathcal{E} &\vdash \text{e}_1 \Rightarrow v_1 \\
  \mathcal{E} &\vdash \{x \mapsto v_1\} \Rightarrow v_2 \\
  \mathcal{E} &\vdash \text{let rec} \; f \; x = \text{e}_1 \; \text{in} \; \text{e}_2 \Rightarrow v
\end{align*}
\]

For addition, conditional and projections the rules are:

\[
\begin{align*}
  \mathcal{E} &\vdash \text{e}_1 \Rightarrow + \\
  \mathcal{E} &\vdash \text{e}_2 \Rightarrow (n_1, n_2) \\
  \mathcal{E} &\vdash \text{e}_1 \; \text{e}_2 \Rightarrow n
\end{align*}
\]

\[
\begin{align*}
  \mathcal{E} &\vdash \text{e}_1 \Rightarrow \text{e}_2 \Rightarrow n_1 \\text{and} \; n_2 \\text{integer} \; \text{and} \; n = n_1 + n_2
\end{align*}
\]

\[
\begin{align*}
  \mathcal{E} &\vdash \text{e}_1 \; \text{e}_2 \Rightarrow n
\end{align*}
\]
\[ \begin{align*}
\mathcal{E} \vdash e_1 \triangleright true & \quad \mathcal{E} \vdash e_2 \triangleright v \\
\mathcal{E} \vdash \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 \triangleright v \\
\mathcal{E} \vdash e_1 \triangleright \text{fst} & \quad \mathcal{E} \vdash e_2 \triangleright (v_1, v_2) \\
\mathcal{E} \vdash (e_1, e_2) \triangleright v_1 \\
\mathcal{E} \vdash e_1 \triangleright \text{snd} & \quad \mathcal{E} \vdash e_2 \triangleright (v_1, v_2) \\
\mathcal{E} \vdash (e_1, e_2) \triangleright v_2
\end{align*} \]

and for parallel operations, the rules are:

\[ \begin{align*}
\mathcal{E} \vdash e_1 \triangleright \text{mkpar} & \quad \mathcal{E} \vdash e_1 \triangleright v \\
\mathcal{E} \vdash e_2 \triangleright \langle e_0, \ldots, e_{p-1} \rangle & \quad \mathcal{E} \vdash \langle v_0, \ldots, v_{p-1} \rangle \\
\mathcal{E} \vdash e_3 \triangleright \langle v_0, \ldots, v_{p-1} \rangle & \quad \forall i \ \mathcal{E} \vdash \langle v_i, \ldots, v_{p-1} \rangle \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_0, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle & \\
\mathcal{E} \vdash e_1 e_2 e_3 \triangleright \langle v_i, \ldots, v_{p-1} \rangle &
\end{align*} \]

where \( \forall i \ f'_i = [x, if \ x = 0 \ then \ v'_0 \ else \ldots \ if \ x = (p-1) \ then \ v'_{p-1} \ else \ nc, \emptyset] \).

\[ \begin{align*}
\mathcal{E} \vdash e_1 \triangleright \langle \ldots, true, \ldots \rangle \\
\mathcal{E} \vdash e_2 \triangleright n \\
\mathcal{E} \vdash e_3 \triangleright v_3 \\
\mathcal{E} \vdash e_1 \triangleright \langle \langle \ldots, false, \ldots \rangle \rangle \\
\mathcal{E} \vdash e_2 \triangleright n \\
\mathcal{E} \vdash e_4 \triangleright v_4 \\
\mathcal{E} \vdash e_1 \triangleright \text{isnc} \\
\mathcal{E} \vdash e_2 \triangleright v \neq \text{nc} \\
\mathcal{E} \vdash (e_1, e_2) \triangleright \text{false} \\
\mathcal{E} \vdash e_1 \triangleright \text{isnc} \\
\mathcal{E} \vdash e_2 \triangleright \text{nc} \\
\mathcal{E} \vdash (e_1, e_2) \triangleright \text{true} \\
\mathcal{E} \vdash \text{nprocs} \triangleright p
\end{align*} \]

4 A BSP Categorical Abstract Machine for BSML

4.0.1 Abstract machines for the \( \lambda \)-calculus

To calculate the values of the \( \lambda \)-calculus, a lot of abstract machines have been designed. The first was the SECD machine [19] which was used for the first implementation of the LISP language. It used environment (a list of values) for the closure and four stacks for the calculus. But it is an old and not optimised machine. In the same spirit, [5] presented the functional abstract machine (FAM). The FAM optimised access to the environment by using arrays (so with a constant cost access). It is to be noticed that for functional languages with a call by name strategy, [28] designed the G-machine with its graph reduction. But we have an eager language so these techniques are not suitable for us.

The CAM, categorical abstract machine, was introduced and used by Curien and Cousineau to implement the CAML language [8] which is a variant of Standard ML [27]. The CAM [12] is an environment machine derived from categorical combinators of Curien [9] and has its roots from equational and denotational semantics of the \( \lambda \)-calculus. A study of this property for BSP computing and BS\( \lambda \)-calculus has been made by [25] but for a static number of processes.

4.0.2 Abstract machines for the BS\( \lambda \)-calculus

For BS\( \lambda \)-calculus, [26] modified the SECD. But this new machine still has the same problems as the original one: slowness, difficulty to have real instruction machine and optimise it, notably for the exchange of closures.

To remedy these problems, [25] introduced a modification of the CAM for BSP 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. Using an abstract machine eases portability but statically defining the number of processors for compilation is against it. Moreover it is not at all adapted for Grid computing.
the instruction of exchange of values is difficult to translate to real code because this instruction adds
instructions to the code during the execution.

The first problem is specific to [25] but the second problem is shared with the BSP SECD machine. We
give here a suitable abstract machine which is an extension of the CAM for BSP computing without these
problems. We first recall the execution model of the original sequential CAM and after its BSP extensions.
Then we explain a technique allowing to compile our core language to this abstract machine in a secure way.

4.1 The BSP CAM

The BSP CAM has two kinds of instructions: sequential and parallel ones. Its corresponds to the two
structures of the original calculus: BSL-calculus [22].

4.1.1 Sequential CAM

The CAM machine is a very simple machine where categorical terms can be considered as code acting on a
graph of values. Instructions are few in number and quite close to real machine instructions. The machine
state has two components:

1. a code pointer \( C \) called program counter representing the code being executed as a sequence of instruc-
tions.
2. a stack \( S \) (a sequence of machine values) called stack pointer holding function arguments, intermediate
results and function return contexts.

Each of them holds a pointer in a real implementation; however, for simplicity, we will describe them as
containing respectively a list of instructions (program counter) and a list of stack elements (stack pointer).
The top of the stack corresponds to the terms (a structured value) which have been computed by the CAM.
It may be viewed as a register. The CAM supposes a de-curryfi ed version of the calculus. This is why all
our operators (and specially the \texttt{apply} operator) use pairs (noted \((s, t)\)).

The values stored in this stack are constants, closures, pairs of semantic values which may in turn be
pairs so that trees may be constructed. The CAM uses closures \([C, s]\) and recursive closures \([C, s]_r\) where
\(C\) is a fragment of CAM code and \(s\) is a value meant to denote an environment for representing functional
values in a natural way since its structure is induced by categorical combinator’s properties (optimize closure
building and environment sharing instead of optimizing access to values). Environments, as in the natural
semantics, are trees of values coding by pairs. Environments give the semantics values of free variables in
closure for the CAM machine which could be used with special instructions. Predefined operations (such as
addition, subtraction ...) may be added to the instructions.

The traditional machine is summarized in Figure 1. It must be read as: \emph{executing the instruction when
the machine is in this state takes it to a new state}. Evaluating a CAM program begins with an empty stack
and ends with a value in the register that is the result of the program and an empty program counter. We
briefly give the meaning of the instructions:

\textbf{Push} : duplicates the register.

\textbf{Swap} : swaps the tops of the stack.

\textbf{Cons} : makes a pair on the tops of the stack.

\textbf{Fst} : expects a term \((s, t)\) and replaces it by \(s\) (resp. \(t\) for the \texttt{Snd} instruction).

\textbf{Cur} : replaces the register \(s\) by the closure \([C, s]\) where \(C\) is in the code encapsulated by the \textbf{Cur}
instruction.

\textbf{CurRec} : replaces the register \(s\) by the recursive closure \([C, s]_r\) where \(C\) is in the code encapsulated by the
\textbf{CurRec} instruction. This is a cyclic closure.

\textbf{App} : expects the register \(\[C, s]\) (resp. \(\[(C, s)], t\)) replaces it by \((s, t)\) (resp. \((s, [C, s]), t\)) prefixed the
rest of the code and the program counter is \(C\).
Quote : replaces the register by the encapsulated constants (int, bool or the special constant of non-communication).

Return : returns the context program counter.

Branch : removes the register and according to whether it is true or false, executed C₁ or C₂.

Add : primitive operator of the addition. Takes a pair on the register and gives the addition of the two components. (Idem for Sub, Equal and another arithmetic operations).

Isnc : primitive operator which tests if the register is the non-communication constant or not.

To represent recursive functions (given by the local recursive binding), some classical solutions exist. The first solution given by the authors [9] of the sequential CAM is a Wind instruction which makes a side effect to simulated cyclic closures (a recursive value where the first slot of the pair points back to the closure itself). Its consists in physically replacing the second value on the top of the stack (supposed to be a pair) by the register and remove it. A second solution given by [6] consists in using recursive environments (a cyclic environment for recursive values where the first slot of the pair corresponding to the recursive variable in the source term, points back to the closure itself) as in their natural semantics [10]. This solution needs to have a new instruction to create this kind of environments and adapt the Fst and Snd instructions. In the same spirit, [7] recursive functions could be done by looping directly in the code instead of cyclic environments in the compilation phases. Usual implementations use the scheme described in [1] to generate real code machine. But, all those solutions are not easy to prove correct w.r.t the dynamic semantics of the source language. In [3] an automatically proof of soundness and correctness of the translation from mini-ML to the CAM and its dynamic semantics have been done with the proof assistant system Coq [2] for mini-ML with recursive closures. We use this solution because in a further work, we will extend this proof to our BSP CAM and prove our machine correct w.r.t. the dynamic semantics. So, we have an instruction Cur for closure and CurRec for recursive one. The recursivity of the closure appears in the App rule when the closure is used again in the environment.

The Branch instruction is used for the if then else (and at) constructor. It was also added to the CAM by [9]. Environments are not mere lists but full binary trees. The categorical combinators Fst and Snd are precisely access functions into those binary trees. The compiling process uses a pattern that is the formal image of the environment. Fst and Snd are also used to represented the fst and snd primitive operators. We refer to [9] and [12] for more details on the CAM and the theory of categorical combinators.

Those instructions which are the traditional instructions of the CAM correspond to the asynchronous steps of the BSP model. In the next section, we add special instructions (which could be seen as categorical combinators) for the parallel machine and for the synchronous steps of the model.

4.1.2 Parallel extensions

To simulate a BSP computer, the BSP CAM is simply a duplicating of the sequential CAM on each process. To express the BSP super-steps we need two kinds of instructions: sequential instructions and synchronous ones. For the first phase of the BSP model (asynchronous calculus), we also need the number of processes and the names of each process (in the spirit of SMPlD programming) given by other sequential instructions:

Nprocs : replaces the register by the constant of the number of processes (p). The actual value is defined at the beginning of execution and not at compilation. Thus a program for our abstract machine is totally portable

Pid : adds on the stack the name of the process.

The Pid 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 CAM: At and Send. They are the only instructions which need BSP synchronization between the sequential CAM 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 CAM machines. We now give the CAM meaning of these instructions:
<table>
<thead>
<tr>
<th>stack</th>
<th>code</th>
<th>stack</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>((t_1, t_2))::S</td>
<td>\text{Fst};C</td>
<td>(t_1))::S</td>
<td>C</td>
</tr>
<tr>
<td>((t_1, t_2))::S</td>
<td>\text{Snd};C</td>
<td>(t_2))::S</td>
<td>C</td>
</tr>
<tr>
<td>(t))::S</td>
<td>\text{Quote}(c);C</td>
<td>c::S</td>
<td>C</td>
</tr>
<tr>
<td>(t))::S</td>
<td>\text{Cur}(C_1);C</td>
<td>[C_1, t]::S</td>
<td>C</td>
</tr>
<tr>
<td>(t))::S</td>
<td>\text{CurRec}(C_1);C</td>
<td>[C_1, t]::S</td>
<td>C</td>
</tr>
<tr>
<td>s::S</td>
<td>\text{Push};C</td>
<td>s::S</td>
<td>C</td>
</tr>
<tr>
<td>(t_1, t_2))::S</td>
<td>\text{Swap};C</td>
<td>t_2, t_1::C</td>
<td>C</td>
</tr>
<tr>
<td>(t_1, t_2))::S</td>
<td>\text{Cons};C</td>
<td>(t_2, t_1))::S</td>
<td>C</td>
</tr>
<tr>
<td>([s_1, s_2, t])::S</td>
<td>\text{App};C</td>
<td>((s, t))::C::S</td>
<td>C</td>
</tr>
<tr>
<td>([s_1, s_2, t])::S</td>
<td>\text{App};C</td>
<td>((s, [C_1, s_2, t])::C::S</td>
<td>C</td>
</tr>
<tr>
<td>t::C_1::S</td>
<td>\text{Return};C</td>
<td>t::S</td>
<td>C_1</td>
</tr>
<tr>
<td>((t_1, t_2))::S</td>
<td>\text{Add};C</td>
<td>((t_1 + t_2))::S</td>
<td>C</td>
</tr>
<tr>
<td>((t_1, t_2))::S</td>
<td>\text{Sub};C</td>
<td>((t_1 - t_2))::S</td>
<td>C</td>
</tr>
<tr>
<td>((t_1, t_2))::S</td>
<td>\text{Equal};C</td>
<td>true::S</td>
<td>C</td>
</tr>
<tr>
<td>((t_1, t_2))::S</td>
<td>\text{Equal};C</td>
<td>false::S</td>
<td>C</td>
</tr>
<tr>
<td>true::S</td>
<td>\text{Branch}(C_1, C_2);C</td>
<td>S</td>
<td>C_1, C</td>
</tr>
<tr>
<td>false::S</td>
<td>\text{Branch}(C_1, C_2);C</td>
<td>S</td>
<td>C_2, C</td>
</tr>
</tbody>
</table>
\[true::S = \text{Branch}(C_1, C_2);C\]
\[false::S = \text{Branch}(C_1, C_2);C\]

Figure 1: Asynchronous instructions

At : replaces the register by the register of the \(n^{th}\) process.

Send : replaces the register, supposed to be a "recursive" pair (see next section) by another pair with exchange of values of pairs between the \(p\) processes.

The instructions of the BSP CAM are given in Figure 2 for a \(p\) processors machine.

Remark: Nproc and Send are the only instructions which depend on the number of processes.

4.2 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 [23] [13] and not De Bruijn indices [4] (they transform an identifier to the number of \(\lambda\)-abstraction which are included between the identifier and the \(\lambda\)-abstraction that binds it; this method was used to solve the problem of binding variables) codings for these; our compiling function will have to deal with the translation of a variable to some access code that will find at run time its value in the environment. Thus the compiling function has an extra parameter which gives the position of the free variables of the expression to be compiled in the environment. We note \(\llbracket e \rrbracket_P\) the function of compilation of an expression \(e\) with the environment \(P\). The compiling function is defined by induction on the expression and begins with an empty environment (). We suppose that the expressions are well-typed and the nested of parallel vectors is rejected by the type checker [14] of the BSML language.

4.2.1 Environments

We have seen that environments are binary trees. So the access function will transform variables on Fst and Snd instructions to access the environment:

\[
\begin{align*}
\llbracket x \rrbracket () &= \text{raise fail} \\
\llbracket x \rrbracket_{(P, x)} &= \text{Snd}; \text{ else raise fail} \\
\llbracket x \rrbracket_{(x, P)} &= \text{Fst}; \text{ else raise fail} \\
\llbracket x \rrbracket_{(P_1, P_2)} &= \text{(Snd; } \llbracket x \rrbracket_{P_2})?((\text{Fst; } \llbracket x \rrbracket_{P_1})
\end{align*}
\]
4.2.2 Sequential mini-BSML expressions

Constants are trivially compiled to Quote and Nprocs instructions:

\[
\begin{align*}
[i]_P & = \text{Quote}(i); \quad \text{if } i \in \mathbb{N} \\
[b]_P & = \text{Quote}(b); \quad \text{if } b \in \mathbb{B} \\
[\text{nc}]_P & = \text{Quote}(\text{nc}); \\
[\text{nprocs}]_P & = \text{Nprocs};
\end{align*}
\]

For the primitive operators we use an extra-function Code\_Operator which gives the instruction of each operator and we have: $[\text{op}]_P = \text{Cur(Snd; Code\_Operator(op)); }$ where:

\[
\begin{align*}
\text{Code\_Operator(op)} = \begin{cases} \\
\text{Add} & \text{where op = +} \\
\text{Sub} & \text{where op = -} \\
\text{Equal} & \text{where op = =} \\
\text{Fst} & \text{where op = fst} \\
\text{Snd} & \text{where op = snd} \\
\text{Isnc} & \text{where op = isnc}
\end{cases}
\end{align*}
\]

Applications, abstractions, pairs, let binding and recursive binding are classically compiled (see [9], [23] [3]). For some optimizations, we use a trick of [9] which compiles differently applications and primitive operators applied to their arguments:

\[
\begin{align*}
\text{Application} & \quad [\text{op} \; e_1 \; e_2]_P = [e_1]_P; \text{Code \_ Operator(op)}; \\
\text{Pair and Abstraction} & \quad [(e_1 \; e_2)]_P = \text{Push; [e_1]_P; Swap; [e_2]_P; Cons; App}; \\
\text{Sequential construction} & \quad [[\text{fun} \; x \to e]]_P = \text{Cur}([e]_P(x); \text{Return}); \\
\text{if e1 then e2 else e3} & \quad \text{Push; [e_1]_P; Branch([e_2]_P; Return; [e_3]_P; Return;);} \\
\text{let rec f x = e_1 in e_2} & \quad \text{Push; CurRec([e_1]_P(P, \text{x}); Return;); [e_2]_P; Cons}; \\
\end{align*}
\]

4.2.3 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. So we add to the Code \_ Operator the two new cases:

\[
\text{Code\_Operator(op)} = \begin{cases} \\
\text{Pid; Cons; App} & \text{where op = mkpar} \\
\text{App} & \text{where op = apply}
\end{cases}
\]

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:

\[
\begin{align*}
\text{if e_1 at e_2 then e_3 else e_4} & = \text{Push; Push; [e_1]_P; Swap; [e_2]_P; At;} \\
& \quad \text{Branch([e_3]_P; Return; [e_4]_P; Return;);}
\end{align*}
\]

The compilation of the put operator is the only real difficulty. To compile the put operator, a first way presented by [25] 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 CAM 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 to directly have the code generated by our compiler:

```plaintext
let put_function = fun f ->
  let rec create_value n = if n=0 then (f n) else ((f n), (create n-1)) in
  let construct_on_case = fun g -> fun pid -> fun value -> fun n -> if n=pid then value
  else (g n) in
  let rec read couple = ( let compteur=(fst couple) in let value=(snd couple) in
  if compteur=0 then (fun n -> if n=0 then value else nc) else
  (((construct_on_case (read (compteur-1,snd value))) compteur) (f value)))
  in read (nproc-1, (create nproc-1));;
```

`create_value` recursively computes the value to send. `read` and `construct_on_case` build recursively the code of the return of the `put` operator. `put_function` is absolutely not a well-typed expression because we simulate lists by recursive pairs. But this is not a problem because the compile phase is done after the type-checker phase.

Now, to have the compilation of 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 of the last pair and the call of the read function:

```plaintext
[put]P = Insere Send([put_function]P)
[put e]P = Push;Insere_Send([put_function]P);Swap;[e]P;Cons;App;
```

**Remark:** All other cases are compiling errors.

### 4.3 Examples

A sequential implementation of the BSP CAM has been made to generate those examples. To illustrate, we compile and run the following trivial expressions (see Figure 2) with two processes:

```plaintext
(mkpar (fun pid -> pid+1));
```

and this one (Figure 3 and Figure 4 for the first process and Figure 5 and Figure 6 for the second one).

```plaintext
if (mkpar (fun i -> if i=(nproc-1) then true else false)) at (nproc-1)
then (mkpar (fun i -> i+2)) else (mkpar (fun i -> i))
```

Where:

- \( C1 = \text{Push;Push;Snd;Swap;Push;Nprocs;Swap;Quote 1;Cons;Sub;Cons;Equal;Branch}(C7,C8);\text{Return;}
- \( C2 = \text{Push;Snd;Swap;Quote 2;Cons;Add;Return;}
- \( C3 = \text{Cur}(C2);\text{Pid;Cons;App;Return;}
- \( C4 = \text{Snd;Return;}
- \( C5 = \text{Cur}(C4);\text{Pid;Cons;App;Return;}
- \( C6 = \text{Swap;Push;Nprocs;Swap;Quote 1;Cons;Sub;At;Branch}(C3,C5;)
- \( C7 = \text{Quote true;Return;}
- \( C8 = \text{Quote false;Return;}
```

**Remark:** the fact that we wrote \( CI \) for the code in the execution of the BSP CAM is near an implementation of this machine. Indeed, program pointer are implemented with "real pointers" and instructions (like \texttt{Cur} or \texttt{Branch}) manipulate those code pointers.
and where i is the name of the process and d the number of processes.

where C: Push and Swap: Quote: Cons::Add:Return.

Figure 2: BSP CAM instructions and BSP CAM example running
Figure 6: The end of the second example for the second process.
5 Conclusions and Future Works

The Bulk Synchronous Parallel Categorical Abstract Machine presented here provides a detailed and portable model of parallel environments management for Bulk Synchronous Parallel ML. It has too advantages with respect to the BSP-SECD machine and the BSP-CAM of [25]:

- 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 with implementation both simpler and more classic.

The next phase in this project should be:
- the proof of correctness of this machine with respect to BSML semantics
- the parallel implementation of this abstract machine.

Acknowledgments

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

References


16


