Pattern Matching of Parallel Values in Bulk Synchronous Parallel ML

Frédéric Dabrowski, Frédéric Loulergue and Frédéric Gava

Laboratory of Algorithms, Complexity and Logic, Créteil, France

`fdabrowski@lspf.org, {loulergue,gava}@univ-paris12.fr`

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. This paper outlines an extension of BSML and of the BSλ-calculus (a calculus of functional bulk synchronous parallel programs) with pattern matching of parallel values.

1. Introduction

Bulk Synchronous Parallelism (BSP) is a parallel programming model introduced by Valiant [11, 10] to offer a high degree of abstraction, in the same way as PRAM models, and yet to allow portable and predictable performance on a wide variety of architectures.

Our previous works embedded the bulk synchronous parallel model into functional languages. We obtained a formal description as an extension of the λ-calculus by a small set of parallel primitives called the BSλ-calculus [9] which is confluent, and a library for the functional programming language Objective Caml [2], called the `BSMLlib` library [8].

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 [4], 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. The Objective Caml compiler is very good and programs which use a lot a dynamic data structures may be more efficient than a program in C (and the Objective Caml program would be much simpler: for example, memory management is automatic). Moreover if a program needs performance improvements it is possible to rewrite parts of it in C or Fortran and then easily embedded them in the Objective Caml program.

Pattern matching is an important feature of high level programming languages. It offers sub-term extraction and function definition by cases. The present work aims at extending the BSλ-calculus and the `BSMLlib` library by pattern matching facilities, especially the matching of parallel values which are distributed tuples of values. The problems with the matching of such values are to define suitable patterns and to performe the matching (which requires communications) as efficiently as possible.

The paper is organized as follows. In section 2 we review the necessary background: the Bulk Synchronous Parallel model, pattern matching in the Objective Caml language and the `BSMLlib` library for functional BSP programming. We then provide in section 3 examples to show how pattern matching of parallel values can be done in the current `BSMLlib` library. The different solutions are not efficient. Thus we conclude the need for an extension of the current library by a new primitive for parallel pattern matching. Section 4 is devoted to the formalization of this new primitive inside a calculus of functional BSP programs. The `BSMLlib` library being a direct extension of Objective Caml, we choose to write a simple calculus which follows the evaluation strategy of this language and which allows the matching of parallel values. Finally we discuss some related work (section 5) and conclude (section 6).

2. Preliminaries

2.1. Bulk Synchronous Parallelism

Bulk-Synchronous Parallelism (BSP) is a parallel programming model introduced by Valiant [11, 10] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. The BSP execution model represents a parallel computation on p processors as a sequence of *super-steps* which are sequences of one computation phase (p asynchronous computations) and one communication phase (data exchanges between processors) ended by one global synchronization. The BSP *cost* model estimates execution times by a simple formula. A computation phase takes as long as its longest sequential process, a global synchronization takes a fixed, system-dependent time L and a communication phase is completed in time proportional to the arity h of the data exchange: the maximal number of words sent or received by a processor during that super-step. The system-dependent constant g, measured in time/word, is multiplied by h to obtain the estimated communication time. It is useful to measure times in multiples of a Flop so as to normalize g and L w.r.t. the sequential speed of processor nodes.

2.2. Pattern Matching in Objective Caml

Objective Caml [2] is a functional language with polymorphic type inference, imperative features, object oriented extensions and a powerful module system. Programs can be compiled to byte-code which can be interpreted by a virtual machine which was ported to a large variety of architectures or to native code. Programs can also be written and run into an interactive top level.

Let us provide two small examples of user defined data types and pattern matching.

Take, as an example, a polymorphic type which adds the value “None” to any existing type:

```
type ’a option = None | Some of ’a
```

This program defines the type `’a option`. A type variable `’a` can be replaced by any type. The right side of the equality enumerates the two *constructors* of this type: the 0-ary constructor `None` and the unary constructor `Some` which takes as argument a value of type `’a`.

If we ask the interactive top level to evaluate the expression `None`, it answers `-: ’a option = None`: the unnamed expression (`-`) has type `’a option` and its value is `None`. If we ask the interactive top level to evaluate the expression `Some (1+1)`, it answers `-: int option = Some 2`: the unnamed expression (`-`) has type `int option` and its value is `Some 2`.

The following function takes two optional integers (type `int option`) and sums them:

```
let optadd v1 v2 = match (v1,v2) with
  | (None,None) -> None
  | (Some v, None) -> Some v
  | (None, Some v) -> Some v
  | (Some v1, Some v2) -> Some (v1+v2)
```

The top level systems answers:

```
val optadd: int option -> int option ->
            int option = <fun>
```

`optadd` has type `int option -> int option -> int option` which is the type of a function which takes two values of type `int option` and returns a value of type `int option` (but it can also be partially applied to only one value. As an illustration `(optadd None)`, the application of the function `optadd` to the value `None`, is a function whose type is `int option -> int option`). The `optadd` function uses pattern matching: the expression `(v1,v2)` is matched against four possible patterns: `(None,None)`, `(Some v,None)`, `(None,Some v)` and `(Some v1,Some v2)`.

The matching is sequential, i.e. if the expression is not matched by the first pattern then the second is tried, etc. In our example the matching is complete: all possible values of the expression can be matched against at least one of the patterns. If one of the cases was removed, the matching would be incomplete: warning messages indicate incomplete matching to the programmer. If a matching is incomplete it is possible that no pattern matches the expression. In this case an exception `Match_failure`, which can be caught, is raised. Patterns must be linear: a variable cannot appear twice in a pattern. This restriction is necessary to have a pattern matching algorithm with linear complexity.

To end with this short presentation, let us write a function which returns the length of a list. The polymorphic and recursive type for lists can be defined by:

```
type ’a list = Nil | Cons of ’a*’a list
```

It is predefined in Objective Caml. The empty list is noted `[]` and to put an element `h` at the beginning of a list `t`, one writes `h::t`. The list $\mathsf{e}_1 :: \mathsf{e}_2 :: \ldots :: \mathsf{e}_n :: []$ can also be written $[\mathsf{e}_1; \mathsf{e}_2; \ldots; \mathsf{e}_n]$. The recursive `length` function can be defined by:

```
let rec length l = match l with
  | [] -> 0
  | h::t -> 1 + length t
```

Its type is `’a list -> int`. A shorter version is:

```
let rec length = fun
  | [] -> 0
  | h::t -> 1 + length t
```

In the following sections we will also use two other patterns: variables and the wild-card pattern _ which matches any value.

2.3. The `BSMLlib` library

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation: a library for Objective Caml [2]. 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 `bsp_p:unit->int` such that the value of `bsp_p()` is p, the static number of processes of the parallel machine. The value of this variable does not change during execution. There is also an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our type system enforces this restriction [5].

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

```
mkpar:  (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. We usually write `f` as `fun pid->e` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*. For example the expression `mkpar(fun i->i)` will be evaluated to the parallel vector $\langle 0, 1, \ldots, p-1\rangle$[1] Note that Objective Caml is an eager language: the arguments are first evaluated, from right to left, then the expression which is applied to the arguments and finally the application is performed. 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`.

One can observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by:

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

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

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

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

The full language would also contain a synchronous conditional operation:

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

such that $\texttt{ifat}(\texttt{v}, \texttt{i}, \texttt{v}_1, \texttt{v}_2)$ will evaluate to $\texttt{v}_1$ or $\texttt{v}_2$ depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par->int->bool` to be used only in the construction: `if (at vec pid) then... else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases. Global conditional is necessary of express algorithms like :

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

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

We end with small examples of functions used in the next section (figure 1). The `get` function takes a parallel vector of values and a parallel vector of integers. Its semantics is given by the following equation:

$$\texttt{get}\langle v_0, v_1, \ldots, v_{p-1}\rangle\langle i_0, i_1, \ldots, i_{p-1}\rangle = \langle v_{i_0}, \ldots, v_{i_{p-1}}\rangle$$

It's parallel cost is $(h+1) \times g + 2 \times L$, where

$$h = \max_{0\leq k<p}\{\sum_{0\leq j<p} |v_j| \text{ such that } i_j = k\}$$

where $|v|$ denotes the size of the value v in words. Experiments showed that the actual performance of `get` follows this cost formula.

3. Parallel Pattern Matching with the `BSMLlib` Library

In the current `BSMLlib` library, pattern matching is possible because it is possible in Objective Caml. Thus one can write:

```
let one = replicate 1
and this = mkpar(fun i->i) in
let l = [one; this] in match l with
 | [] -> None
 | h::t -> Some h
```

[1].

```
let replicate x = mkpar(fun pid ->x)
let parfun f v = apply (replicate f) v
let get vec srcs =
  let pids = parfun (fun i->natmod i (bsp_p())) srcs in
  let ask = put(parfun (fun i dst->if dst=i then Some() else None) pids)
  and replace_by_data =
    parfun2 (fun f d dst->match(f dst)with Some() -> Some d|_->None) in
  let reply = put(replace_by_data ask vec) in
  parfun (fun(Some x)->x) (apply reply pids)
```

Figure 1. Examples

In this case, the matching of a parallel value (`l` is a parallel value since it is a list of parallel vectors) is possible because the patterns match only the structure of the list (which is the same at each process) and could be applied to any kind of list.

We can also use pattern matching on a parallel value in applying a usual Objective Caml pattern at each process:

```
parfun length (mkpar(fun i->l_i))
```

In this case the matching cannot change the global behavior of the program because the matching is done asynchronously on each process, on usual sequential values. There is no global agreement to determine which local pattern was chosen.

When one writes programs using the `BSMLlib` library, one would like to be allowed to write pattern matching of parallel values in the two following cases.

First the matching of parallel vectors at the global level, with the possibility to have patterns which depend on the internal structure of parallel values. Take, as an example, a usual Objective Caml pattern P. $\langle P\rangle$ would be a pattern which matches a parallel vector whose components are sequential values which may be matched by P. Of course the right side of a $\langle P\rangle$`->  E` must be in this case a parallel expression, otherwise the nesting of parallel vectors would be allowed.

We refer to [5] for an in-depth discussion of the reasons why such nesting should be avoided and for a polymorphic type system which enforces this restriction. In particular in the `BSMLlib` library, the programmer is responsible for the absence of nesting of parallel vectors (or he or she must use our type checker). Two rules to avoid such nesting are:

- one should not have functions which return a value of a usual Objective Caml type and which take arguments with parallel type.
- one should not write local binding expressions `let` $\mathtt{e}_1 = \mathtt{e}_2$ `in` $\mathtt{e}_3$ where $\mathtt{e}_2$ is a parallel value and $\mathtt{e}_3$ is a usual Objective Caml value.

Another interesting pattern matching of parallel values would be the global matching of parallel vectors where the patterns may be different on each processor. The problem is:

- to find a syntax to express such patterns;
- to have a syntax to manipulate the values bound during pattern matching.

The latter facility is an open problem and we believe that there is no simple solution to allow such pattern matching and that a solution would be too complicated to be used in practice. Thus we address only the former in this paper.

The general mechanism is perhaps best elucidated by examples. In this parallel case we match at the global level but looking the internal structure of parallel vectors, so it is possible that a pattern matches the expression at some process but not at some others:

```
type number = F of float | I of int
let v1 = mkpar(fun pid->if (pid/2) = 0
            then I pid
            else F (float_of_int pid))
let f1 e = match e with
  <I i> -> <i>
| <F f> -> replicate 0
```

`v1` is an expression of type `number par` and `f1` of type `number par -> int par`. No pattern of our example program `f1` matches expression `v1` because on some processes the local value held by `v1` is matched sometimes by the pattern `F f`, sometimes by `I i`. Thus communications are needed to determine whether a $\langle P\rangle$ pattern matches an expression or not. Each process must check locally the pattern matching and then say to one chosen process (for example process 0) if the pattern matches the expression or not. Then a `if at` construction is used to change the global behavior.

Take as an example the small following program:

```
let f2 e = match e with
  | <Some v> -> (<v>,<v>)
```

In the pattern `<Some v>` the variable `v` has type `int` and represents several sequential values, one per process. Thus it is not allowed to use `v` in the right side of the matching. The correct notation is `<v>` which is a parallel vector of values.

The `f2` program can be seen as syntactic sugar for the `BSMLlib` program:

```
let f2' e =
 let local_match e =
  match e with
     |Some v -> true |_ -> false
 and extract_v e =
  match e with |Some v -> v in
 let vl=gl 0 (parfun local_match e) in
 let vb = parfun (reduce (&&)) vl in
 if at vb 0 then
 let vv = parfun extract_v e in (vv,vv)
 else raise Parallel_Match_failure
```

where `gl` has type `int -> 'a par -> 'a list par` and `gl i vv` gathers the values of parallel vector `vv` at process `n`, and whose semantics is given by:

$$\texttt{gl}\ \mathtt{n}\ \langle v_0, \ldots, v_1\rangle = \langle\ [\,]\ , \ldots ,\ \underbrace{[\mathtt{v_0}; \ldots; \mathtt{v_{p-1}}]}_{n}\ , \ldots ,\ [\,]\ \rangle$$

and `reduce` $\oplus[\mathtt{v_1}; \ldots; \mathtt{v_n}] = [\mathtt{v_1}; \mathtt{v_1} \oplus \mathtt{v_2}; \ldots; \oplus_{\mathtt{k=1}}^{\mathtt{k=n}} \mathtt{v_k}]$.

In the sequential case, we said in section 2.2, that each pattern is tried until one pattern matches the given expression. If we take the same strategy in the parallel case, each try costs two BSP super-steps which is too expensive. It is possible to reduce this cost to one super-step but in this case the program is not well typed: it is possible implement the new `match with` construct as a *primitive* but it is no more syntactic sugar.

Another possibility is to first check locally all the patterns then exchange the booleans which indicate whether a pattern matches the expression or not and determine the first j such as the j^{th} pattern matches the expression on all processes. For example:

```
let f3 e = match e with
  | <Some v> -> <v>
  | <None>   -> (replicate 0)
```

is syntactic sugar for the program given in figure 2.

This program is again not very efficient since we must use as many `if at` as patterns. Thus for n patterns, the cost would be in the worst case, the cost of the total exchange $n \times (p-1) \times g + L$ plus the cost of each `if at`: $n \times ((p-1) \times g + L)$. To improve the efficiency of such a program to make the number of super-steps independent on the number of patterns, we could introduce a new primitive in the language: a `match e at n with` construction whose parallel cost would be the same as a `if at`. The end of the `f3'` function would then became:

```
match vj at 0 with
 | 0 -> let extract_v e =
        match e with | Some v -> v in
    let vv = parfun extract_v e in vv
 | 1 -> replicate 0
 | _ -> raise Parallel_Match_failure
```

and cost would be: $n \times (p-1) \times g + L + (p-1) \times g + L$. This new construction is a straightforward generalization of the `if at` construct and it will be included in the next release of the `BSMLlib` library.

It can be make even more efficient but no more as syntactic sugar. One can notice that the j value is a parallel vector which contains the same value everywhere. Thus as in the implementation of the `BSMLlib` [8] the type `' a par` is defined as `'a`, in the implementation of the core library, the following program, which uses the usual pattern matching of Objective Caml, would be correct (but it is an incorrect user program, because it would not be well typed):

```
match vj with
 | 0 -> let extract_v e =
        match e with | Some v -> v in
    let vv = parfun extract_v e in (vv)
 | 1 -> replicate 0
 | _ -> raise Parallel_Match_failure
```

The communication and synchronization cost of this last version would be $n \times (p-1) \times g + L$.

In conclusion of this section, the matching of parallel values using the new parallel pattern $\langle P\rangle$ where P is a usual Objective Caml pattern must be implemented as a *primitive* of the `BSMLlib` library in order to attain reasonable performance. Designing it as syntactic sugar on top on the current `BSMLlib` library would lead to poor performances.

Each times we add a new primitive to our language, we design a formal semantics to ensure the correct behavior of this primitive by proving the confluence of the new semantics. That is what is done in the next section.

4. BSλ-calculus with Pattern Matching of Parallel Values

In the λ-calculus, the notion of function relies on the abstraction of a variable in a term as in $\lambda x.x + x$ where λx means the variable x is abstracted in the term $x + x$. The application $((\lambda x.e_0)\ e_1)$ denotes a computation step done by substituting the argument e_1 to the variable x in e_0. Pattern matching allows a more general abstraction mechanism through the use of nested patterns, generally a subset of terms whom variables bound free occurences as in the λ-calculus. Take, as an exemple, terms $\lambda(x, y).x$ and $\lambda(x, y).y$ the usual projections.

```
let f3' e =
 let local_checkP1 e = match e with | Some v -> true | _ -> false
 and local_checkP2 e = match e with | None -> true | _ -> false in
 let apply_check e = map (fun f->f e) [local_checkP1;local_checkP2] in
 let l1 =parfun apply_check e in  let l2 = total_exchange l1 in
 let vj = let rec aux j l =
  if j=bsp_p() then raise Parallel_Match_failure
  else let h,l' = split(map (fun (h::l)->h,l) l) in
       let b = reduce (&&) h in
       if b then j else aux (j+1) l' in
  parfun (aux 0) l2 in
 if at (parfun (fun x ->x=0) vj) 0 then
   let extract_v e = match e with |Some v -> v in let vv=parfun extract_v e in vv
 else if at (parfun (fun x ->x=1) vj) 0 then replicate 0
 else raise Parallel_Match_failure
```

$$\text{where} \begin{cases} \texttt{map}\,\mathtt{f}\,[\mathtt{e}_1;\ldots;\mathtt{e}_\mathtt{n}] & = & [\mathtt{f}\,\mathtt{e}_1;\ldots;\mathtt{f}\,\mathtt{e}_\mathtt{n}] \\ \texttt{split}\,[(\mathtt{e}_1,\mathtt{e}'_1);\ldots;(\mathtt{e}_\mathtt{n},\mathtt{e}'_\mathtt{n})] & = & ([\mathtt{e}_1;\ldots;\mathtt{e}_\mathtt{n}],[\mathtt{e}'_1;\ldots;\mathtt{e}'_\mathtt{n}]) \\ \texttt{total\_exchange}\,\langle\, e_0\,,\ldots,\,e_{p-1}\,\rangle & = & \langle\,[e_0;\ldots;e_{p-1}]\,,\ldots,\,[e_0;\ldots;e_{p-1}]\,\rangle \end{cases}$$

Figure 2. `f3'` **program**

4.1. Syntax

Let $\dot{x}, \dot{y}, \dot{z}, \ldots$ range over the set $\dot{V}$ of local variables and $\bar{x}, \bar{y}, \bar{z}, \ldots$ range over the set $\bar{V}$ of global variables. $\dot{c}$ ranges over a set of constants : integers or booleans. Finally, we use the uppercase letters $C, D, E, \ldots$ to denote constructors. We define $\dot{p}$ and $\dot{e}$ (resp. $\bar{p}$ and $\bar{e}$), sets of local (resp. global) patterns and terms of the BSλ-calculus. $\dot{v}$ (resp. $\bar{v}$) denotes the set of local (resp. global) values.

$$\begin{array}{rcl}
\dot{p} & ::= & \_ \;\mid\; \dot{c} \;\mid\; \dot{x} \;\mid\; (\dot{p},\dot{p}) \;\mid\; C\,\dot{p} \\
\dot{a} & ::= & \lambda\dot{p}[\dot{e_1}].\dot{e_0} \;\mid\; (\lambda\dot{p}[\dot{e_1}].\dot{e_0}|\dot{a}) \\
\dot{e} & ::= & \dot{x} \;\mid\; \dot{c} \;\mid\; (\dot{e},\dot{e}) \;\mid\; \dot{e}\;\dot{e} \;\mid\; \dot{a} \\
\dot{v} & ::= & \dot{c} \;\mid\; (\dot{v},\dot{v}) \;\mid\; \dot{a} \\
\bar{p} & ::= & \_ \;\mid\; \bar{x} \;\mid\; (\bar{p},\bar{p}) \;\mid\; (\bar{p},\dot{p}) \;\mid\; (\bar{p},\dot{p}) \\
 & \mid & p@e \;\mid\; \langle\dot{p}\rangle \;\mid\; C\,\bar{p} \\
\bar{a_1} & ::= & \lambda\dot{p}[\dot{e}].\bar{e} \;\mid\; (\lambda\dot{p}[\dot{e}].\bar{e}|\bar{a_1}) \\
\bar{a_2} & ::= & \lambda\bar{p}[\bar{e}].\bar{e} \;\mid\; (\lambda\bar{p}[\bar{e}].\bar{e}|\bar{a_2}) \\
\bar{e} & ::= & \bar{x} \;\mid\; \langle\dot{x}\rangle \;\mid\; \bar{e}\;\bar{e} \;\mid\; \bar{e}\;\dot{e} \\
 & \mid & \texttt{mkpar}\,e \;\mid\; \texttt{get}\,\bar{e}\,\bar{e} \;\mid\; \texttt{apply}\,\bar{e}\,\bar{e} \\
 & \mid & \langle\dot{e},\ \dot{e},\ \ldots,\ \dot{e}\rangle \;\mid\; \texttt{if}\,\bar{e}\,\texttt{at}\,\dot{e}\,\texttt{then}\,\bar{e}\,\texttt{else}\,\bar{e} \\
 & \mid & (\bar{e},\bar{e}) \;\mid\; (\bar{e},\dot{e}) \;\mid\; (\dot{e},\bar{e}) \;\mid\; \bar{a_1} \;\mid\; \bar{a_2} \\
\bar{v} & ::= & (\bar{v},\bar{v}) \;\mid\; (\bar{v},\dot{v}) \;\mid\; (\dot{v},\bar{v}) \;\mid\; \bar{a_1} \;\mid\; \bar{a_2}
\end{array}$$

We will use p and e to denote either local or global patterns and terms. Intuitively, the term in brackets in an abstraction denotes a condition which must evaluates to $\mathbf{T}$ in order to allow reduction. We name *delocalized pattern* a pattern of the form $\langle p\rangle$, these patterns are used to translate the local structure of vectors to the global level. In λ-calculus, one can only abstract on a single variable. Patterns generalize the abstraction to more complicated structures which may contain several variables. Thus we have to define the set of variables of a pattern. The variable in $Var(p)$ are the variables bounded by the abstraction:

$$\begin{array}{rcl}
\multicolumn{3}{c}{Var(\_) = \emptyset \quad Var(\dot{c}) = \emptyset \quad Var(x) = \{x\}} \\
Var((p_1,p_2)) & = & Var(p_1) \cup Var(p_2) \\
Var(C\ p) & = & Var(p) \\
Var(\dot{p}@e) & = & Var(\dot{p}) \\
Var(\langle\dot{p}\rangle) & = & \{\langle\dot{x}\rangle / \dot{x} \in Var(\dot{p})\}
\end{array}$$

We also define the set of free variable of a term e and the substitution of a variable x by an expression e in a term by induction (figure 3).

4.2. Semantics

The semantics is given in figure 4, where

$$\gamma_{\alpha\beta} = \begin{cases} \sigma_1 \cup \sigma_2 \text{ if } \alpha = \sigma_1,\ \beta = \sigma_2 \\ \bot_\mathrm{m}\ \text{ if } \alpha \text{ or } \beta = \bot_\mathrm{m} \end{cases}$$

and $\alpha' \in \{\sigma, \bot_m\}$. We first define rules for the `match` operator which maps a set of couple $\{(e_1,x_1),\ldots,(e_n,x_n)\}$ to a pattern p and a expression e such as $p[e_1/x_1,\ldots,e_n/x_n] = e$, if such a set exists. The matching rules return $\bot_m$ otherwise. These rules acts as usual except for patterns of the form $\langle\dot{p}\rangle$ for which the local structure of vectors is translated to global level as mentioned above and patterns of the form $\dot{p}@e$ which are use to get global control on the matching of a parallel value component. For example, if the pattern is $\langle(\dot{p_1},\dot{p_2})\rangle$ and the parallel vector holds value (v_i, v'_i) at process i, the result is the union of the matchings of $\langle\dot{p_1}\rangle$ with the parallel

$$
\begin{array}{rcl}
\multicolumn{3}{c}{fv(x) = x \quad fv(\langle \dot{x} \rangle) = \langle \dot{x} \rangle \quad fv(\dot{c}) = \emptyset} \\
fv(C_i\ p) & = & fv(p) \\
fv((p_1, p_2)) & = & fv(p_1) \cup fv(p_2) \\
fv(\lambda p[e_1].e_0) & = & (fv(e_1) \cup fv(e_2)) \backslash Var(p) \\
fv((\lambda p[e_1].e_0\ \ |a)) & = & (fv(\lambda p[e_1].e_0) \cup fv(a)) \\
fv(e_1\ e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(\mathtt{mkpar}\ \dot{e}) & = & fv(\dot{e}) \\
fv(\langle e_0,\ e_1,\ \ldots,\ e_{p-1} \rangle) & = & \emptyset \\
fv(\mathtt{apply}\ \bar{e_1}\ \bar{e_2}) & = & fv(\bar{e_1}) \cup fv(\bar{e_1}) \\
fv(\mathtt{get}\ \bar{e_1}\ \bar{e_2}) & = & fv(\bar{e_1}) \cup fv(\bar{e_2}) \\
fv\left(\begin{array}{l} \mathtt{if}\ \bar{e_0}\ \mathtt{at}\ \dot{e} \\ \mathtt{then}\ \bar{e_1}\ \mathtt{else}\ \bar{e_2} \end{array} \right) & = & \bigcup_{i=0,1,2} fv(\bar{e_i}) \cup fv(\dot{e})
\end{array}
$$

$$
\begin{array}{lcllcl}
x\{e/x\} & = & e & x\{e/y\} & = & x \text{ if } y \neq x \\
\langle \dot{x} \rangle\{e/\langle \dot{x} \rangle\} & = & e & \langle \dot{x} \rangle\{e/\langle \dot{y} \rangle\} & = & \langle \dot{x} \rangle \text{ if } y \neq x \\
x\{e/\langle y \rangle\} & = & x & \langle x \rangle\{e/y\} & = & \langle x \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle e_0,\ e_1,\ \ldots,\ e_{p-1} \rangle\{e/x\} & = & \langle e_0,\ e_1,\ \ldots,\ e_{p-1} \rangle \\
\langle e_0,\ e_1,\ \ldots,\ e_{p-1} \rangle\{e/\langle \dot{x} \rangle & = & \langle e_0,\ e_1,\ \ldots,\ e_{p-1} \rangle \\
(\lambda p[e_1].e_0)\{e/x\} & = & (\lambda p[e_1\{e/x\}].e_0\{e/x\}) \\
 & & \text{if } x \notin Var(p) \\
(\lambda p[e_1].e_0)\{e/\langle \dot{x} \rangle\} & = & (\lambda p[e_1\{e/\langle \dot{x} \rangle\}].e_0\{e/\langle \dot{x} \rangle\}) \\
 & & \text{if } \langle \dot{x} \rangle \notin Var(p) \\
(\lambda p[e_1].e_0|a)\{e/x\} & = & ((\lambda p[e_1].e_0)\{e/x\}|a\{e/x\}) \\
(\lambda p[e_1].e_0|a)\{e/\langle \dot{x} \rangle\} & = & ((\lambda p[e_1].e_0)\{e/\langle \dot{x} \rangle|a\{e/\langle \dot{x} \rangle\})
\end{array}
$$

Figure 3. Definitions

vector which holds the value v_i at process i and $\langle \dot{p_2} \rangle$ with the parallel vector which holds the value v'_i at process i. In the second case, the matching is as usual but with a given component from a parallel value and the result is available at each process. Next, we define the reduction rules of the BSλ-calculus with pattern matching.

A simple abstraction acts as usual except that we need to match the pattern and the argument. The reduction condition must evaluate to T, after substitution, in order to allow reduction of the body where the substitution is applied. If the match operator returns $\perp_{\mathrm{m}}$ then the expression does not reduce. A compound abstraction $(\lambda p.e_0|a)\ \ e_1$ is reduced to $e_0\{x_1/e_1, \ldots, e_n/x_n\}$ where $\{(e_1, x_1), \ldots, (e_n, x_n)\}$ is the result, if exists, of the match operator applied to p and e. If match returns $\perp_m$ then it is reduced to $(a\ e')$.

The parallel operators behave as indicated by their informal presentation in section 2.3.

The reduction system is fully deteministic since on the one hand, for a given rule only one rule applies and on the other hand, the matching rules hold the same property.

5. Related Work

To our knowledge the first work on the formalization of pattern matching (in the sequential case) was done by V. van Oostrom [12]: he defined the $\lambda\phi$-calculus, an untyped λ-calculus extended with pattern matching and studied its confluence. The patterns are a subset of the terms, but the confluence holds only if this set satisfies the "rigid pattern condition". [6] starts from the assignment of patterns and terms to Gentzen's sequent proofs. The operational semantics of this typed calculus is then obtained as the computational interpretation of proofs using the Curry-Howard isomorphism.

Those extensions are only for pure λ-calculus, but in functional programming languages, there are built-in basic types (for e.g. booleans, integers, floats). These basic types and the primitives operations on them can be formalized using rewriting rules. The interaction of rewriting systems and λ-calculus leading to higher order rewriting systems has been widely studied. [3] presents the extension of such a family of systems, ERS (expression reduction systems), with pattern matching.

To our knowledge there is no work on the matching of parallel values because either the languages use implicit parallelism, so pattern matching is as usual, or the languages with explicit parallelism do not offer data types for parallel values but rather follow the SPMD programming style, so pattern matching of parallel values is not supported.

6. Conclusions and Future Work

This papers demonstrates that the current `BSMLlib` library and its formalization the BSλ-calculus are not suitable for the pattern matching of parallel values. We proposed a new calculus of functional Bulk Synchronous Parallel programs which allows efficient pattern matching of parallel values. This confluent calculus is the basis of an extension of the `BSMLlib` library with facilities for parallel pattern matching. The implementation of such an extension is sketched.

Currently, we transform the matching of parallel values $\langle P \rangle$ by hand, but we are writing an automatic transformation using the preprocessing facilities of the Objective Caml compiler. The automatic transformation follows exactly the process described in this paper.

The semantics presented here can lead to irreductible terms which are not values when `match` returns $\perp_m$. In an actual language, we need to be able to catch such a failure. Thus we need an exception handling mechanism. In the current `BSMLlib` library difficulties come when an exception raised inside a `mkpar` for example is not caught at the local level: the whole program fails. We can imagine to catch such exceptions at the global level. We are exploring the different possible ways to give a meaning to programs such as:

$$(m_1)\frac{}{\textbf{match}(x,\ v) \to_m \{(v,\ x)\}} \quad (m_2)\frac{}{\textbf{match}(\langle \dot{x} \rangle,\ v) \to_m \{(v,\ \langle x \rangle)\}} \quad (m_3)\frac{}{\textbf{match}(\_,\ v) \to_m \emptyset}$$

$$(m_4)\frac{}{\textbf{match}(\dot{c},\ \dot{c}) \to_m \emptyset} \quad (m_5)\frac{\dot{c} \neq v}{\textbf{match}(\dot{c},\ v) \to_m \bot_m} \quad (m_6)\frac{\textbf{match}(p_1,\ v_1) \to_m \alpha \quad \textbf{match}(p_2,\ v_2) \to_m \beta}{\textbf{match}((p_1,\ p_2),(v_1,\ v_2)) \to_m \gamma_{\alpha\beta}}$$

$$(m_7)\frac{\textbf{match}(p,\ v) \to_m \alpha'}{\textbf{match}(E\ p,\ E\ v) \to_m \alpha'} \quad (m_8)\frac{v \neq Cv'}{\textbf{match}(E\ p,\ v) \to_m \bot_m} \quad (m_9)\frac{\textbf{match}(\dot{p},\ v_j) \to_m \alpha'}{\textbf{match}(\langle \dot{p}@j \rangle,\ \langle v_0, v_1, \ldots v_{p-1} \rangle) \to_m \alpha'}$$

$$(m_{10})\frac{\forall i \in \{0,\ 1,\ \ldots,\ p-1\}.\textbf{match}(\dot{p},\ v_i) \to_m \sigma^i}{\textbf{match}(\langle \dot{p} \rangle,\ \langle \dot{v_0},\ \dot{v_1},\ \ldots,\ \dot{v_{p-1}} \rangle) \to_m \sigma} \quad \sigma(\langle x \rangle) = \langle \sigma^0(x),\ \sigma^1(x),\ \ldots,\ \sigma^{p-1}(x) \rangle$$

$$(m_{11})\frac{\exists i \in \{0,\ 1,\ \ldots,\ p-1\}.\textbf{match}(\dot{p},\ v_i) \to_m \bot_m}{\textbf{match}(\langle \dot{p} \rangle,\ \langle \dot{v_0},\ \dot{v_1},\ \ldots,\ \dot{v_{p-1}} \rangle) \to_m \bot_m}$$

$$(1)\frac{e_2 \to v \quad \textbf{match}(p,\ v) \to \sigma \quad e_1\sigma \to \mathbf{T} \quad e_0\sigma \to v_0}{(\lambda p[e_1].e_0)e_2 \to v_0} \quad (2)\frac{e_2 \to v \quad \textbf{match}(p,\ v) \to \sigma \quad e_1\sigma \to \mathbf{T} \quad e_0\sigma \to v_0}{(\lambda p[e_1].e_0|a)e_2 \to v_0}$$

$$(3)\frac{e_2 \to v \quad \textbf{match}(p,\ v) \to \sigma \quad e_1\sigma \to \mathbf{F} \quad ae_2 \to v_0}{(\lambda p[e_1].e_0|a)e_2 \to v_0} \quad (4)\frac{e_2 \to v \quad \textbf{match}(p,\ v) \to \bot_{\mathrm{m}}}{(\lambda p[e_1].e_0|a)e_2 \to ae_2} \quad (5)\frac{}{v \to v} \quad (6)\frac{e \to v}{E\ e \to E\ v}$$

$$(7)\frac{e_1 \to v_1 \quad e_2 \to v_2}{(e_1, e_2) \to (v_1, v_2)} \quad (8)\frac{\bar{e_1} \to \langle \dot{f_0},\ \dot{f_1},\ \ldots,\ \dot{f_{p-1}} \rangle \quad \bar{e_1} \to \langle \dot{v_0},\ \dot{v_1},\ \ldots,\ \dot{v_{p-1}} \rangle \quad \forall i \in \{0,\ 1,\ \ldots,\ p-1\}.\dot{f_i}\ \dot{v_i} \to \dot{w_i}}{\bar{e_1}\ \#\ \bar{e_2} \to \langle \dot{w_0},\ \dot{w_1},\ \ldots,\ \dot{w_{p-1}} \rangle}$$

$$(9)\frac{\forall i \in \{0,\ 1,\ \ldots,\ p-1\}.(\dot{e}\ i) \to \dot{w_i}}{\pi\ \dot{e} \to \langle \dot{w_0}, \dot{w_1}, \ldots, \dot{w_{p-1}} \rangle} \quad (10)\frac{\bar{e_1} \to \langle \dot{v_0},\ \dot{v_1},\ \ldots,\ \dot{v_{p-1}} \rangle \quad \bar{e_2} \to \langle \dot{m_0},\ \dot{m_1},\ \ldots,\ \dot{m_{p-1}} \rangle \quad \forall i \in \{0,\ 1,\ \ldots,\ p-1\}.m_i \in \mathcal{N}}{\bar{e_1}\ ?\ \bar{e_2} \to \bar{e_1} \to \langle \dot{v_{m_0}},\ \dot{v_{m_1}},\ \ldots,\ \dot{v_{m_{p-1}}} \rangle}$$

Figure 4. Semantics

```
try mkpar(function
 |0->raise Exn1 |1->raise Exn2 |_-> i)
with ...
```

Exceptions handling will increase the safety of our language, in particular we will investigate this both practically and theoretically in order to check the applicability of type systems which can detect uncaught exceptions in Caml programs [7].

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

7. References

[1] A. Asperti and C. Laneve. Interactive systems: the theory of optimal reduction. *Mathematical Structures in Computer Science*, 4:457–504, 1994.

[2] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000. freely available in english at http://caml.inria.fr/oreilly-book.

[3] J. Forest and D. Kesner. Expression Reduction Systems with Patterns. In *14th International RTA Conference*, volume 2706 of *LNCS*, pages 107–122. Springer-Verlag, 2003.

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

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

[6] Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, January 1996.

[7] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM TOPLAS*, 22(2):340–377, 2000.

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

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

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

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

[12] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Vrije Universiteit, Amsterdam, 1990.