UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

An Introduction to BSλ

Frédéric Loulergue
Gaétan Hains
Université d'Orléans, LIFO

Rapport N^o **98-09**

An Introduction to BSλ

Frédéric Loulergue and Gaétan Hains
{loulergu,ghains}@lifo.univ-orleans.fr

January 4, 1999

Abstract

This paper is an introduction to BSλ a calculus of parallel-recursive BSP programs. The calculus is presented, the confluence of its flat subset is stated and examples are given in an ML-like syntax.

1 Introduction

If parallel programming is to become as widespread as sequential programming, the languages supporting it should incorporate all the standard abstraction mechanisms including higher-order functions and recursion. Yet for such languages to be practical scalable programming tools, abstraction should not come at the expense of predictable performance. Unfortunately many parallel languages don't describe data placement so that performance is not predictable as a function of the source program. This problem occurs when data placement depends on the language implementation, instead of its semantics.

On the contrary, Valiant's bulk synchronous parallel (BSP) computing paradigm [Val90, McC96] demonstrates that programs explicitly written for a static number p of processors can have predictable execution costs on a wide variety of architectures. But the combination of BSP algorithms with high-level language features is not well understood so that the evolution of BSP languages is hindered. We are investigating this combination from a functional programming perspective and introduce here an extension of the λ-calculus with BSP operations. The *flat* part of the calculus (formalizing BSML[Lou98]) is inspired by the paradigm of algorithmic skeletons [Col89]; it is obtained by adding constants with special rules to the λ-calculus. But it is more restrictive than the standard notion of δ-rules[Bar86] in the following way: terms are divided into two sorts. One sort describes global objects and the other describes usual "sequential" objects, prevented from containing global terms. This absence of sort-nesting is verified statically in the syntax and realizes the concept of *explicit processes* by making explicit the data-parallel structure of terms. Explicit processes are intended to facilitate the construction of cost measures for BSP execution.

Despite this practical advantage, the flat calculus can not express divide-and-conquer parallel algorithms[Gor96] and thus complicates the task of writing explicit messages. We address this problem by extending our flat calculus with a synchronous parallel composition used for the recursive definition of bulk synchronous parallel operations.

In our calculus, the reduction of parallel composition to data-parallel operations is given a formal description suggesting a *dynamic* realization. Moreover, the explicit processes in the flat calculus together with the BSP cost model[Ski96] offer the possibility of performance prediction from the recursive parallel program. Unfortunately, only the flat subset of the calculus has been found to be confluent and is thus a correct basis for equational-inductive program proofs or for the design of functional languages.

In the next section we present the BSλ calculus. The next section presents the confluence of BSλ. We then discuss reduction strategies and illustrate the calculus with two program for the same BSP algorithm, one flat and one recursively parallel.

2 The calculus

We now describe the BSλ syntax and its equational theory with denotational and/or operational motivations. The reader is assumed to be familiar with the elements of BSP [Ski96].

2.1 Syntax

We consider a set $\dot{V}$ of *local* variables and a set $\bar{V}$ of *global* variables. Let $\dot{x}$, $\dot{y}$, ... denote local variables and $\bar{x}$, $\bar{y}$, ... denote global variables from now on. x will denote a variable which can be either local or global.

The syntax of BSλ begins with *local* terms e: λ-terms representing programs or values stored in a processor's local memory. The set $\dot{\Lambda}$ of local terms is given by the following grammar:

$$e \ ::= \ \dot{x} \ \mid \ e\,e \ \mid \ \lambda\dot{x}.\,e \ \mid \ c$$

where $\dot{x}$ denotes an arbitrary local variable. We will abbreviate to $(e_1 \to e_2, e_3)$ the conditional term $e_1\ e_2\ e_3$ where the value of e_1 is β-equivalent to the Church encoding of T or F. The boolean operation $\Rightarrow$ (needed in rule $(\|\pi)$ below) is also assumed to be encoded as a λ-term[Bar86].

The principal BSλ terms E are called *global* and represent data fields i.e. maps from a fixed set of processor names to values. The exact processor names are left unspecified but assumed to be (closed) λ-terms so that data fields can be expressed intensionally by a special constant called π (for *parallel* object). For illustrations purposes we let the processor names be 0,. . .,p-1 (e.g. Church numerals) and draw data fields as p-vectors of values. The term $\pi\, e$ will represent a data field whose values are given by function e applied to the processors names(Figure 1).

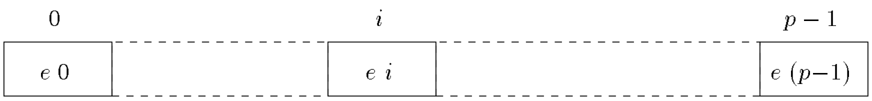

Figure 1: The term πe

The set $\bar{\Lambda}$ of global terms is given by the following grammar[1]:

$$\begin{array}{rcl} E & ::= & \bar{x} \ \mid \ E\,E \ \mid \ E\,e \ \mid \ \lambda\bar{x}.\,E \ \mid \ \lambda\dot{x}.\,E \\ & \mid & \pi\ e \ \mid \ E \# E \ \mid \ E\,?\,E \ \mid \ (E \overset{e}{\to} E, E) \ \mid \ E \parallel_e E \end{array}$$

and has the following denotational meaning. Let p be the *finite* number of processor names $\mathcal{N} = \{n_i \mid \ 0 \le i < p\}$, each being a closed λ-term. Global terms denote maps from $\mathcal{N}$ to local values,

[1]Terms of the form $(\lambda\bar{x}.E)e$ (resp. $(\lambda\dot{x}.E)E'$) are not β-contracted and constitute implicit "errors" because they present a local argument to a global$\to$global function (resp. a global argument to a local$\to$global function). In practice a type system should eliminate them, but we will not discuss this here.

functions between them ($\lambda\bar{x}.\ E$) or functions from local values to such maps ($\lambda\dot{x}.\ E$). In particular the denotation of $\pi\ e$ has for value at n_i the value (normal form) of $e\ n_i$. The forms $E_1 \# E_2$ and $E_1 \,?\, E_2$ are called parallel application (*apply-par*) and *get* respectively. Apply-par represents point-wise application of a field of functions to a field of values, i.e. the pure computation phase of a BSP superstep. Get represents the communication phase of a BSP superstep: a collective data exchange with a barrier synchronization. In $E_1 \,?\, E_2$, the resulting data field contains values from E_1 taken at processor names defined in E_2. The exact meanings of apply-par and get are defined by the BSλ equations. The last two forms of global terms define synchronous conditional expressions and synchronous parallel composition. The meaning of $(E_1 \stackrel{e}{\rightarrow} E_2, E_3)$ (not to be confused with $sie_1e_2e_3$ is that of E_2 (resp. E_3) if the data field denoted by E_1 has value T (resp. F) at the processor name denoted by e. The meaning of $E_1 \parallel_e E_2$ at n_i is the meaning of E_1 applied to n_i if $e\ n_i = T$ and is the meaning of E_2 applied to n_i if $e\ n_i = F$. In other words, $E_1 \parallel_e E_2$ is a partition of the network into independent parts. We will see that the two parts are also required to synchronize together so that their joint effect remains a BSP computation.

2.2 Equational semantics

We now define the equality theory of BSλ terms. The equality of local terms is simply β-equivalence: the reflexive-transitive and symmetric closure of

$$(\beta) \qquad (\lambda\dot{x}.\ e)e' = e[\dot{x} \leftarrow e']$$

to any subterm (and possibly of rules for the local conditional).

The equality of global terms is defined by syntax-related rules and context rules which determine the applicability of the former. We now give the syntax-related rules.

First, there are axioms for global beta-equivalence.

$$\begin{array}{llcl} (B) & (\lambda\bar{x}.\ E)E' & = & E[\bar{x} \leftarrow E'] \\ (B') & (\lambda\dot{x}.\ E)e' & = & E[\dot{x} \leftarrow e'] \end{array}$$

There are also axioms for the interaction of the field constructor π with the other BSP operations.

Equations $(?\pi)$ (Figure 2), $(\#\pi)$ (Figure 3) and $(\|\pi)$ (Figure 4) encode the denotational meaning of the BSP operations on fields. In particular, get is functional composition inside the data field construction. The value of $\pi\, e_1 \,?\, \pi\, e_2$ at processor name n_i is $e_1\,(e_2\, n_i)$, i.e. the value of $\pi\, e_1$ at processor name $e_2\, n_i$. Notice that, in practical terms, this represents an operation whereby every processor receives one and only one value from one and only one other processor. This restriction can be lifted but will be retained here for the sake of simplicity.

$$(?\pi) \quad (\pi\ e_1) \,?\, (\pi\ e_2) \quad = \quad \pi(\lambda\dot{x}.\ e_1(e_2\ \dot{x}))$$

$$(\#\pi) \quad (\pi\ e_1)\#(\pi\ e_2) \quad = \quad \pi(\lambda\dot{x}.\ (e_1\ \dot{x})(e_2\ \dot{x}))$$

$$(\|\pi) \quad (\pi\ e_1) \parallel_e (\pi\ e_2) \quad = \quad \pi\ (\lambda\dot{x}.\ (e\ \dot{x} \rightarrow e_1\ \dot{x}, e_2\ \dot{x}))$$

Next, the global conditional is defined by two rules whose numerators refer to local computations.

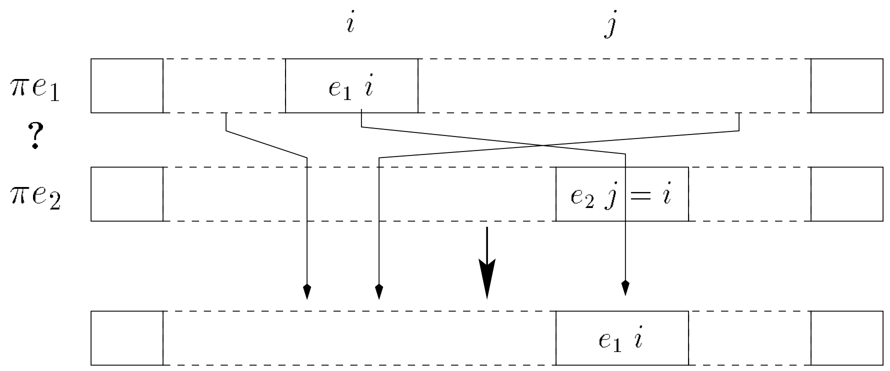

Figure 2: The rule $(?\pi)$

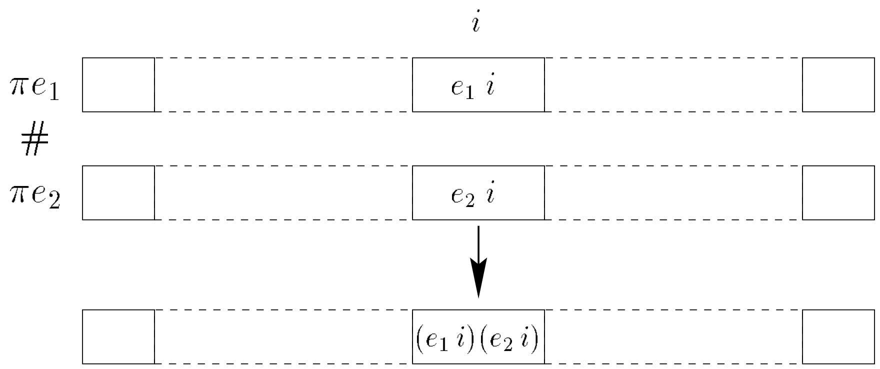

Figure 3: The rule $(\#\pi)$

$$(\stackrel{e}{\to} T) \qquad \frac{e\ e' \doteq T \quad e' \in \mathcal{N}}{((\pi\ e) \stackrel{e'}{\to} E_1, E_2) \ = \ E_1}$$

$$(\stackrel{e}{\to} F) \qquad \frac{e\ e' \doteq F \quad e' \in \mathcal{N}}{((\pi\ e) \stackrel{e'}{\to} E_1, E_2) \ = \ E_2}$$

The above two rules (Figure 5 for the first case) apply outside the context of parallel compositions and generate the following bulk-synchronous computation: first a pure computation phase where all processors evaluate local term e' yielding value v; then processor v evaluates $e\ v$ into value v'; if $v' = T$ (resp. F) then processor v broadcasts the order for global evaluation of E_1 (resp. E_2); otherwise the computation fails.

The remaining rules define the interaction of parallel composition with apply-par, get and the global conditional. They determine a symmetric mode of execution whose recursion trees are balanced with respect to (the global) communication phases [2].

[2]The reader interested in the notion of recursive BSP computations will find motivating remarks in [Hai98].

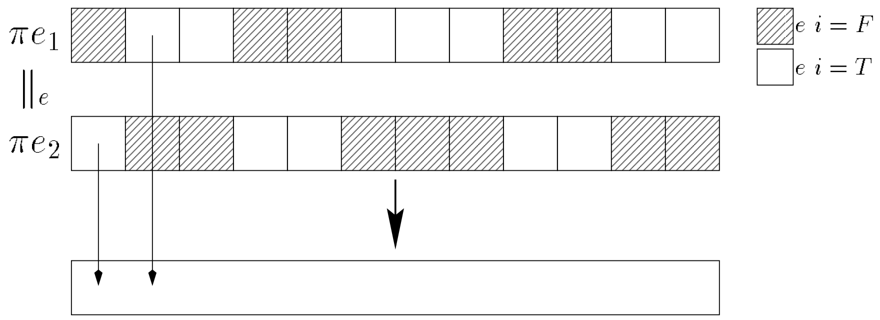

Figure 4: The rule $(\|\pi)$

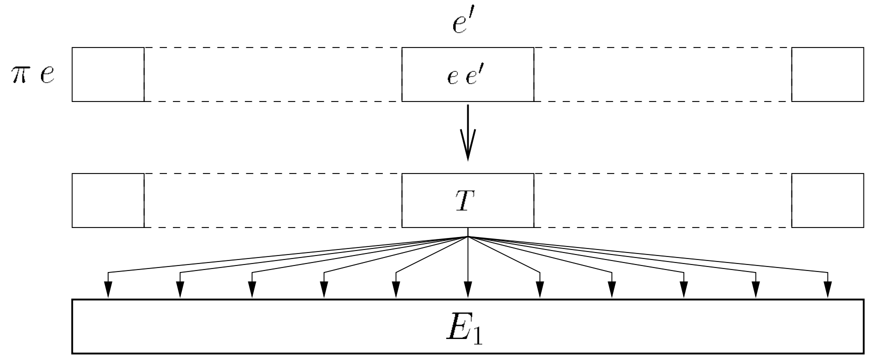

Figure 5: The rule $(\stackrel{e}{\to} T)$

Parallel composition commutes with apply-par because it involves no communication or synchronization (Figure 6).

$$(\|\#) \qquad (E_1 \# E_2) \ \|_e\ (E'_1 \# E'_2) \ = \ (E_1 \ \|_e\ E'_1) \# (E_2 \ \|_e\ E'_2)$$

The rule for parallel composition of two get operations is a conditional commutation law (Figure 7). If applied from left to right it reduces a parallel composition to a get operation.

$$(\|?) \quad \frac{\forall i \in \mathcal{N} \quad \begin{array}{c}(\Rightarrow (e\ i)(e(e_2\ i))) \doteq\ T \\ \text{and} \\ (\Rightarrow (e(e'_2\ i))(e\ i)) \doteq\ T\end{array}}{(E_1\,?\,(\pi\ e_2)) \ \|_e\ (E'_1\,?\,(\pi\ e'_2)) \ = \ (E_1 \ \|_e\ E'_1)\,?\,((\pi\ e_2) \ \|_e\ (\pi\ e'_2))}$$

The syntactic form of $(\|?)$ forces both branches of the parallel composition to be get operations. In this manner the two sides of a parallel compositions must generate an equal number of get operations so as to constitute a unified sequence of global communication phases. The p preconditions of $(\|?)$ are to be realized in parallel over all processors evaluating the term. They are sufficient to ensure that messages are kept within the sub-network where they originate; in other words if E generates a get request for data at a location i where $e\ i \doteq F$ then the program $E \ \|_e\ E'$

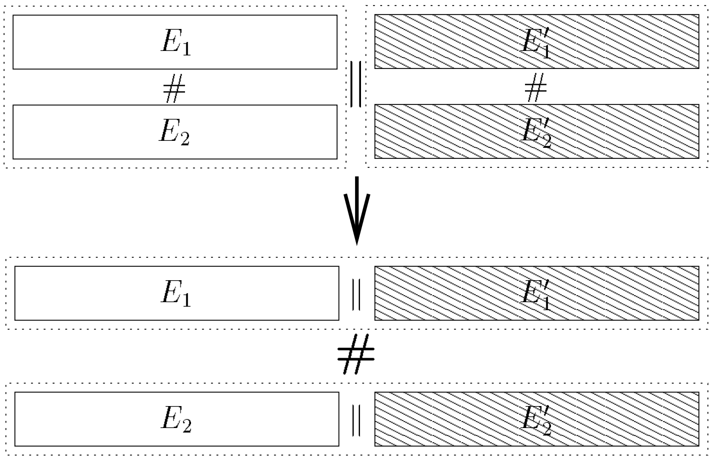

Figure 6: The rule $(\|\#)$

will abort while trying to execute this rule. The following is an informal explanation of how this condition is encoded in the numerator of rule $(\|?)$.

Suppose $(\Rightarrow (e\ i)(e(e_2\ i))) \doteq T$ holds for every $i \in \mathcal{N}$. Then for one particular i the definition of $\Rightarrow$ implies

$$e\ i \doteq A \quad \text{and} \quad (e(e_2\ i))) \doteq B \tag{1}$$

where $A, B \in \{T, F\}$ and $B \doteq T$ if $A \doteq T$. We know that A, value of $e\ i$, is the value of the selector for $(\ldots) \|_e (\ldots)$ at processor i. Similarly B, value of $(e(e_2\ i))$, is the value of the selector at the processor whose name is the value of $e_2\ i$ which is the value of right-hand-side of the left get operation $(\pi\ e_2)$ at processor i. It follows that B is the value of the selector at the processor whose name is indicated by $(\pi\ e_2)$ at i. Condition (1) therefore means: if processor i is concerned with the left part $E_1\,?\,(\pi\ e_2)$ of the parallel composition (it makes the selector e true) then so is the processor from which it requests a value in this get operation. The other condition in the numerator of rule $(\|?)$ means (by contraposition) that if processor i is concerned with the right part of the parallel composition (it makes the selector e false) then so is the processor from which it requests a value in $E'_1\,?\,(\pi\, e'_2)$.

The last BSλ rules allow branches of a parallel composition to use conditional expressions. Given the parallel composition of a global conditional with another one, it is first checked that each synchronizing processor term (e_2 or e'_2) indicates a processor in the appropriate part of the network and then the value of the condition data field (e_1 or e'_1) is evaluated at that processor location. If the result is T then the first branch of the conditional is taken for this part of the network.

$$(\| \to TT) \qquad \frac{\begin{array}{ccc} a\ e_2 \doteq T & e_1\ e_2 \doteq T & e_2 \in \mathcal{N} \\ a\ e'_2 \doteq F & e'_1\ e'_2 \doteq T & e'_2 \in \mathcal{N} \end{array}}{((\pi\ e_1) \overset{e_2}{\to} E_3, E_4)\ \|_a\ ((\pi\ e'_1) \overset{e'_2}{\to} E'_3, E'_4) \;=\; E_3\ \|_a\ E'_3}$$

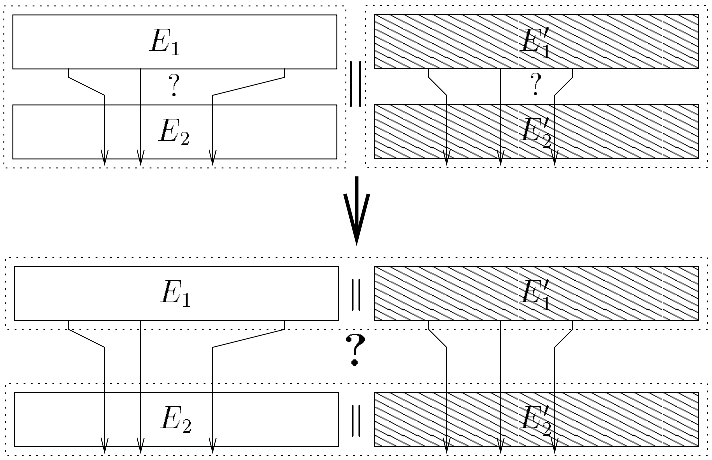

Figure 7: The rule $(\|?)$

Similarly if the value is F, the second branch is taken

$$(\| \to FT) \quad \frac{\begin{array}{ccc} a\ e_2 \doteq T & e_1\ e_2 \doteq F & e_2 \in \mathcal{N} \\ a\ e'_2 \doteq F & e'_1\ e'_2 \doteq T & e'_2 \in \mathcal{N} \end{array}}{((\pi\ e_1) \stackrel{e_2}{\to} E_3, E_4)\ \|_a\ ((\pi\ e'_1) \stackrel{e'_2}{\to} E'_3, E'_4)\ =\ E_4\ \|_a\ E'_3}$$

and there are symmetrical rules for the remaining two possibilities.

$$(\| \to TF) \quad \frac{\begin{array}{ccc} a\ e_2 \doteq T & e_1\ e_2 \doteq T & e_2 \in \mathcal{N} \\ a\ e'_2 \doteq F & e'_1\ e'_2 \doteq F & e'_2 \in \mathcal{N} \end{array}}{((\pi\ e_1) \stackrel{e_2}{\to} E_3, E_4)\ \|_a\ ((\pi\ e'_1) \stackrel{e'_2}{\to} E'_3, E'_4)\ =\ E_3\ \|_a\ E'_4}$$

$$(\| \to FF) \quad \frac{\begin{array}{ccc} a\ e_2 \doteq T & e_1\ e_2 \doteq F & e_2 \in \mathcal{N} \\ a\ e'_2 \doteq F & e'_1\ e'_2 \doteq F & e'_2 \in \mathcal{N} \end{array}}{((\pi\ e_1) \stackrel{e_2}{\to} E_3, E_4)\ \|_a\ ((\pi\ e'_1) \stackrel{e'_2}{\to} E'_3, E'_4)\ =\ E_4\ \|_a\ E'_4}$$

2.3 Context rules

The context rules of BSλ are numerous but easily described negatively as follows. All equations are applicable in any context except for $(?\pi)$, $(\#\pi)$, $(\stackrel{e}{\to} T)$ and $(\stackrel{e}{\to} F)$ which are not applicable in the context of a parallel composition $\|_e$. The reasons for these restrictions are that get interacts correctly with parallel composition by rule $(\|?)$ and that the global conditional, when placed under a parallel composition, performs its choices under control of the composition's selector in rules $(\| \to TT)$, $(\| \to FT)$, $(\| \to TF)$ and $(\| \to FF)$. The context of parallel composition defines the data vs. control parallelism boundary. Outside of it, the flat data-parallel calculus applies. Inside of it, special rules like $(\|?)$ apply to yield a (seemingly) control-parallel calculus.

3 Confluence of the flat calculus

If we orient from left to right the equalities of the previous section, we obtain a reduction relation which is not confluent in general.

The reduction relation for the flat calculus (without parallel composition) written $\rightarrow$, is however confluent.

Theorem 1 *$\forall M, M_1, M_2 \in \bar{\Lambda}$ such that $M \rightarrow^* M_1$ and $M \rightarrow^* M_2$ there exists $M_3 \in \bar{\Lambda}$ such that $M_1 \rightarrow^* M_3$ and $M_2 \rightarrow^* M_3$:*

$$\begin{array}{ccc} & M & \\ \swarrow^* & & \searrow^* \\ M_1 & & M_2 \\ \searrow^* & & \swarrow^* \\ & M_3 & \end{array}$$

This property makes flat BSλ a suitable basis for functional BSP languages where the data-processor mapping and communications are explicit (the so-called *direct mode* of BSP programming). It also ensures the correctness (modulo normalization) of program-equivalence proofs built from BSλ equations.

4 Reduction strategy

Let us now define the reduction strategy for a programming language based on BSλ . The strategy defines the left-to-right application of our rules. For the same reasons that motivate ML — interaction with side effects, imperative extensions to the language — and because of the higher complexity of lazy evaluation we favor [Hai94] call-by-value weak β-reduction for the local terms and the same strategy for (B) and (B') on global terms. In this reduction mode, $e_1\, e_2\, e_3$ cannot serve as conditional because the Church encoding on which it is based is not effective in weak reduction. Constants and δ-rules must therefore be added to the local λ-calculus for boolean operations including the conditional.

- None of the 14 reduction rules is applicable in the context of a local or global λ-abstraction.
- All rules are applicable within the right subterm of a local or global application ee, Ee, EE.
- All rules are applicable within the left subterm of a local or global application ev, Ev, EV whose right subterm is a value: a weakly-normalized term.
- The exact reduction order of the two subterms in an apply-par $E \# E'$ or get $E\,?\,E'$ operation is irrelevant at the current level of description. All rules are applicable within the subterms of such expressions.
- No rules are applicable within the branches E_1, E_2 of a global conditional $(E \overset{e}{\rightarrow} E_1, E_2)$. All the rules are applicable within E and all relevant rules are applicable within e.
- The reduction of the two subterms in a parallel composition $E_1 \parallel_e E_2$ is assumed *not* to be ordered. As in the equational theory, all rules except $(?\pi)$, $(\overset{e}{\rightarrow} T)$ and $(\overset{e}{\rightarrow} F)$ are applicable in E_1 or E_2.

Having defined this strategy we recall that the e in expression $\pi\, e$ is meant to be a function from processor names to values, as in $\pi(\lambda\dot{x}.\, e')$. We then observe that the e' in $\pi(\lambda\dot{x}.\, e')$ will in general contain x as free variable. As a result this subterm e' cannot use δ-rules: they must be applied to ground terms if Mitschke's theorem is to apply. But δ-rules are needed for two reasons:

1. They allow the introduction of external "opaque" functions.
2. They allow the definition of the conditional when using weak reduction.

As a result, reduction in BSλ appears too weak to become the basis of a programming language. We tackle this problem by invoking the intended models for BSλ , namely data-field spaces. If $\mathcal{N}$, the set of processor names has finite cardinality p, then equalities of the form

$$\pi\, e \;=\; \pi\, e'$$

should be proved by p proofs in the λ-calculus:

$$\forall i \in \mathcal{N}.\; e\; i = e'\; i$$

but since p is a priori unbounded, a complete proof of $\pi\, e \;=\; \pi\, e'$ should cover all values of i as follows. So to reduce local terms in the scope of a π we use a term $\mathrm{case}_{\mathcal{N}}$ which is such that:

$$\mathrm{case}_{\mathcal{N}}\; v_1\; \ldots\; v_n\; e\; i \;=\; \begin{cases} v_i & \text{if } i \in \mathcal{N} \\ e\; i & \text{if } i \notin \mathcal{N} \end{cases}$$

and introduce a new "extensional" reduction rule:

$$(\pi)\quad \frac{\forall i \in \mathcal{N}.\;\; e\; i \;\triangleright\; v_i}{\pi\; e \;\triangleright\; \pi\; (\mathrm{case}_{\mathcal{N}}\, v_1\; \ldots\; v_n\; e)}$$

The term $\mathrm{case}_{\mathcal{N}}$ exists because $\mathcal{N}$ is a finite set of ground terms[Bar86]. If term e above is forced to be closed, then rule (π) is a δ-rule and does not affect confluence or normalization.

5 Examples

To illustrate the expressive power of BSλ as programming language, we have programmed in it the standard binary scan algorithm which sum the prefixes of a field with p elements with a binary operation in $\log_2 p$ supersteps. There are two possible version of the algorithm in BSλ : one using only the flat language and the other using parallel recursion (Figure 11). For those examples, written in a ML-like syntax (Figure 8),

the processor names are the Church numerals **0,1,. . .,p-1**.

The flat fold program **scan** (Figure 9) is a linear recursion as shown by the single occurrence of **scan_aux** in the expression defining the function **scan_aux**.

The recursive parallel scan program **scan_rec** (Figure 10) is a binary recursion interleaved with parallel composition as shown in the expression defining the function **scan_aux**. The advantage of a parallel recursive language is illustrated by the following difference between the two programs. In **scan**, address and data-related computation (**cond, msg, local**) is completely disjoint from control (**scan_aux**). As a result the program is relatively difficult to write. On the contrary, **scan_rec** mixes address and data-related computation with its main control term **scan_aux**. This kind of definition allows the programmer to reason by induction on p, a more intuitive design/proof method.

When `nprocs=4` the reduction on figure 12 can be obtained.

$\lambda\bar{x}.\,E$	`fun` $X \rightarrow E$
$\lambda\dot{x}.\,E$	`fun` $x \rightarrow E$
$\lambda\dot{x}.\,e$	`fun` $x \rightarrow e$
$\pi(\lambda x.\,e)$	`par` (`fun` $x \rightarrow e$)
$E_1 \# E_2$	`apply` E_1 `to` E_2
$E_1\ ?\ E_2$	`get` E_1 `from` E_2
$(E \stackrel{e}{\rightarrow} E_1, E_2)$	`if` E `at` e `then` E_1 `else` E_2
$E_1 \parallel_e E_2$	`where` e `do` E_1 `elsewhere` E_2
p	`nprocs`

Figure 8: ML-like syntax

```
let scan op X =
 let rec scan_aux n X =
  let cond j n = (j>=n) in
   let msg n=par(fun j -> if (cond j n) then j-n else j)
   and local n=par(fun j -> if (cond j n ) then op else fun x y -> x in
   if par(fun j-> n>=nprocs) at 0
   then X
   else scan_aux (n*2) (apply (apply (local n) to X) to (get X from (msg n))
 in scan_aux 1 X ;;
```

Figure 9: scan: flat program

6 Conclusions

There has been many attempts at combining functional programming with parallel architectures and it is not possible to review them in any serious manner here for lack of space. Nevertheless the integration of the BSP paradigm and its explicit processes with functional programming has not been successfully realized, at least to our knowledge. Some related languages are Skil [BK96] and Caml Flight [FC95]. The former is embedded in an imperative language and combined with efficient compilation of higher-order functions, it attains very high levels of performance. The latter transforms the pure-functional subset of Caml into a deterministic language with possible interleaving of asynchronous data-parallel operations. Also Skil places constraints on the higher-order functions that can be applied; and in Caml Flight, the environment-preserving expression constructor sync(e) is dynamically prevented from nesting. By comparison, BSλ places no constraint on functions and prevents π-nesting statically.

Being an equational theory, BSλ offers a possible basis for BSP program equivalence proofs. For the confluent flat BSλ calculus, such proofs can be entirely inductive/equational as with any pure functional language. As additional motivation of BSλ , it offers a new presentation of the duality: microscopic view vs macroscopic view in data-parallel semantics [Bou91].

The relationship of control parallelism to data parallelism is internal to the BSλ theory and is possibly its most important property. Unfortunately the introduction of the parallel composition operator breaks confluence so this relationship cannot yet be studied in its full generality within

```
let scan_rec op X =
 let rec scan_aux a b X =
  let msg a b = par(fun j -> ((a+b)/2))
  and local a b = par(fun j -> if (((a+b)/2)<j & j<=b) then op else fun x y -> x) in
  if par(fun j->a=b) at 0
  then X
  else let G Y = apply (apply (local a b) to (get Y from (msg a b))) to Y in
        G (where (fun j->a<=j&j<=((a+b)/2)) do scan_aux a ((a+b)/2) X
           elsewhere scan_aux (1+((a+b)/2)) b  X
 in scan_aux 0 (nprocs-1) X;;
```

Figure 10: scan: parallel recursive program

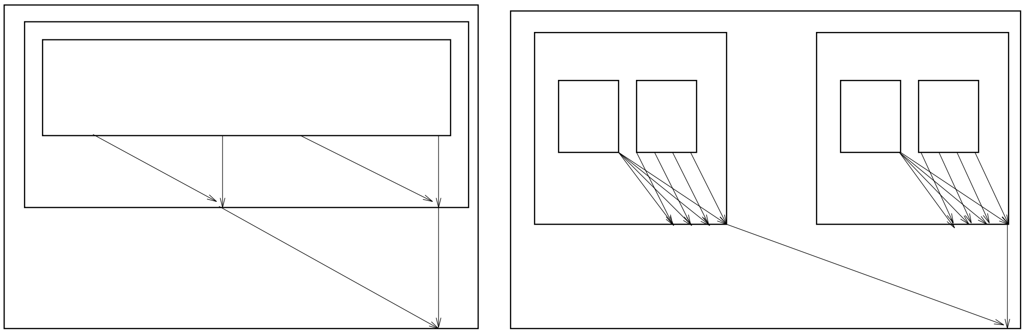

Figure 11: scan programs: flat and parallel recursive. p=4. Processor space is horizontal, time (supersteps) is vertical

functional programming theory. Despite this problem flat BSλ is, in our opinion, the most general framework for the definition of generic skeletons languages [Col89]. In all previous language definitions the introduction of algorithmic skeletons amounts to the addition of δ-rules [DFH$^+$93] or puts constraints on the source language's recursion [CMMP97] or higher-order functions [BK96].

Future work on flat BSλ must address the following problems: an evaluation semantics should be defined and a distributed execution semantics derived from it. An exact BSP cost prediction model and an implementation can then follow.

$$\left.\begin{array}{r}\texttt{scan}\\ \texttt{scan\_rec}\end{array}\right]\texttt{(prefix +)(par (fun j ->j))}$$

$$\equiv$$

$$\texttt{par(fun j->match j with 0->0|1->1|2->3|3->6|x->x)}$$

Figure 12: Example of reduction for nprocs=4

Acknowledgements

Christian Foisy has collaborated to the definition of BSλ. This work was supported by a graduate scholarship of the French MENER (F.Loulergue). Luc Bougé has kindly shared with the second author an earlier attempted formalization [BP91] of the xapping concept [SH86]. BSλ was also inspired by David Skillicorn's miniBSP concept [SHM96]. Arnaud Lallouet has suggested the current form of π. Our earlier version of the formalism was less general: it used terms of the form $\pi\, x.\, e$ where x is bound in e.

References

[Bar86] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1986.

[BFMR96] L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Euro-Par'96 Parallel Processing*, number 1123–1124 in Lecture Notes in Computer Science, Lyon, August 1996. LIP-ENSL, Springer.

[BK96] G.-H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In Bougé et al. [BFMR96].

[Bou91] L. Bougé. On the semantics of languages for massively parallel SIMD architectures. In E. H. L. Aarts and J. van Leeuwen, editors, *PARLE '91, Parallel Architectures and Languages Europe*, number 505 and 506 in Lecture Notes in Computer Science, Eindhoven, June 1991. Springer.

[BP91] L. Bougé and C. Paulin. Incomplete technical report on CMML. 1991.

[CMMP97] C.B.Jay, M.I.Cole, M.Sekanina, and P.Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par'97*, number 1300 in LNCS. Springer, 1997.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[DFH+93] J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.

[FC95] C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[Gor96] S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In Bougé et al. [BFMR96].

[Hai94] G. Hains. Parallel functional languages should be strict. In B. Perhson and I. Simon, editors, *Workshop on General Purpose Parallel Computing. World Computer Congress*, volume 1, pages 527–532. IFIP, North-Holland, September 1994.

[Hai98] G. Hains. Subset synchronization in BSP computing. In H.R Arabnia, editor, *PDPTA'98, International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 242–246, Las Vegas, July 1998. CSREA Press.

[LHF] F Loulergue, G. Hains, and C. Foisy. A Calculus of Recursive-Parallel BSP Programs. Extended version with proof of confluence. Submitted for publication.

[Lou98] F. Loulergue. BSML : Programmation BSP purement fonctionnelle. In D. Méry and G.-R. Perrin, editors, *Dixième Rencontres Francophones du Parallélisme (Renpar'10)*, pages 243–246, Strasbourg, june 1998.

[McC96] W.F. McColl. Universal computing. In Bougé et al. [BFMR96].

[SH86] G.H. Steele and W.D. Hillis. Connection Machine Lisp: A Fine Grained Parallel Symbolic Processing Language. In *ACM Conf. on Lisp and Funct. Progr.*, pages 276–297, Cambridge MA, 1986.

[SHM96] David B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. Technical Report 15-96, Programming Research Group, Oxford University Computing Laboratory, August 1996.

[Ski96] David B. Skillicorn. miniBSP: A BSP Language and Transformation System. Departement of Computing and Information Science, Queen's University, Kingston, Canada, October 1996.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

Publications récentes du LIFO
LIFO's recent publications

97-7 E. MELIN, B. RAFFIN, X. REBEUF, B. VIROT
A Unifying Approach of Data and Task Parallelism Based on Structural Clock Communication Paradigm (extended version)

97-8 S. ANANTHARAMAN, Editor
Proceedings of UNIF'97, International Workshop on Unification

97-9 F. BENTAYEB, D. LAURENT
Inversion de l'algrèbre relationnelle et mises à jour

97-10 I. PARFENOFF
An efficient parallel algorithm for maximum matching in P_4-tidy graphs

97-11 F. ROUSSEL, I. RUSU
An O(m2+mn) algorithm to recognize Meyniel graphs

97-12 M. BAHMA, G. HAINS
Prevision des performances de la θ-jointure et de la multi-jointure pour des machines SN

97-13 F. LOULERGUE, G. HAINS
Design of an event-oriented functionnal parallel language : the finite case

97-14 S. LIMET, F. SAUBION
A proof method for logic programming

97-15 E. MELIN, B. RAFFIN, X. REBEUF, B. VIROT
An Automatic Distribution of Sequential Code Fitting Irregular Data Accesses

Pour récupérer les rapports de recherche du LIFO / To get LIFO's research reports :

- Par WWW : http://www.univ-orleans.fr/SCIENCES/LIFO/Recherche/Publis/RR/index.html
- Par FTP : ftp://ftp-lifo.univ-orleans.fr/pub/RR
- Par courrier papier adressé à / By surface mail adressed to :
  Arnaud Lallouet, LIFO, Université d'Orléans
  4, rue Léonard de Vinci, BP 6759
  F-45067 Orléans cedex 2, FRANCE
  en mentionnant votre adresse.
- Par courrier électronique à / By e-mail to : Arnaud.Lallouet@lifo.univ-orleans.fr