Multi-ML: Programming
Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava and Julien Tesson

Laboratory of Algorithmic Complexity and Logic
Université Paris-Est

BSPSP October 2015
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Multi-ML</td>
</tr>
<tr>
<td>3</td>
<td>Results</td>
</tr>
<tr>
<td>4</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction
   - OCAML
   - BSML
   - MULTI-BSP

2. Multi-ML

3. Results

4. Conclusion
Ocaml: a ML language

Strengths of Ocaml

- A functional programming language

V. Allombert et al.

BSPSP 2015
Ocaml: a ML language

Strengths of Ocaml

- A functionnal programming language
- A powerful type system
Ocaml: a ML language

Strengths of Ocaml

- A functionnal programming language
- A powerful type system
- User-definable algebraic data types and pattern matching
Ocaml : a ML language

Strengths of Ocaml

- A functionnal programming language
- A powerful type system
- User-definable algebraic data types and pattern matching
- Automatic memory management
Ocaml: a ML language

Strengths of Ocaml

- A functionnal programming language
- A powerful type system
- User-definable algebraic data types and pattern matching
- Automatic memory management
- Efficient native code compilers
Syntaxe overview

```ocaml
# let f = fun x → "Hello_" ^ (string_of_int x) in
  let lst = [0;1;2;3] in
  List.map f lst;;
- : string list = ["Hello_0"; "Hello_1"; "Hello_2"; "Hello_3"]

# let pair = ([0;1;2;3],true);;
val pair : int list * bool = ([0; 1; 2; 3], true)

# type 'a list =
  Nil
  | Node of 'a*'a list ;;
type 'a list = Nil | Node of 'a * 'a list
```
Bulk Synchronous ML

What is BSML?

• Explicit BSP programming with a functional approach
• Based upon ML implemented over OCaml
• Formal semantics → computer-assisted proofs (Coq)
What is BSML?

- Explicit BSP programming with a functional approach
What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
- Formal semantics → computer-assisted proofs (COQ)
What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
- Formal semantics $\rightarrow$ computer-assisted proofs (COQ)

Main idea

Parallel data structure $\Rightarrow$ vectors:
BSML primitives

Asynchronous primitives

- $\langle e, \ldots, e \rangle$
- $v$: on processor $i$, assumes $v = \langle v_0, \ldots, v_{p-1} \rangle$
- $\text{pid}$: A predefined vector: $i$ on processor $i$

Synchronous primitives

- $\text{proj}$: $\langle x_0, \ldots, x_{p-1} \rangle \mapsto \langle \text{fun} \ i \to x_i \rangle$
- $\text{put}$: $\langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle \text{fun} \ i \to f_i(0), \ldots, \text{fun} \ i \to f_i(p-1) \rangle$
BSML primitives

Asynchronous primitives

- $\langle e \rangle$ : $\langle e, \ldots, e \rangle$
BSML primitives

Asynchronous primitives

- \( \ll e \gg : \langle e, \ldots, e \rangle \)
- \( v \) : \( v \) on processor \( i \), assumes \( v \equiv \langle v_0, \ldots, v_{p-1} \rangle \)
BSML primitives

Asynchronous primitives

- \[ \ll e \rr : \langle e, \ldots, e \rangle \]
- $v$: $v_i$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
- $\text{pid}$: A predefined vector: $i$ on processor $i$
BSML primitives

Asynchronous primitives

- $\ll e \gg : \langle e, \ldots, e \rangle$
- $v_i$ : $v$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
- $\text{pid}$ : A predefined vector: $i$ on processor $i$

Synchronous primitives

- $\text{proj} : \langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i)$
BSML primitives

Asynchronous primitives

- \(\ll e \gg \) : \(\langle e, \ldots, e \rangle\)
- \(\$v\$ \) : \(v_i\) on processor \(i\), assumes \(v \equiv \langle v_0, \ldots, v_{p-1} \rangle\)
- \(\$\text{pid}\$\) : A predefined vector: \(i\) on processor \(i\)

Synchronous primitives

- \(\text{proj} \) : \(\langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i)\)
- \(\text{put} \) : \(\langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle (\text{fun } i \rightarrow f_i \ 0), \ldots, (\text{fun } i \rightarrow f_i \ (p-1)) \rangle\)
For a BSP machine with 3 processors:

```ocaml
# let vec = """Hello"""" ;;
val vec : string par = <"Hello", "Hello", "Hello">  
# let vec2 = $vec$^(string_of_int $pid$) ;;
val vec2 : string par = <"Hello0", "Hello1", "Hello2">  
# let totex v = List.map (proj v) procs;;
val totex : 'a Bsml.par → 'a list = <fun>  
# totex vec2;;

- : string list = ["Hello0"; "Hello1"; "Hello2"]
```
The **MULTI-BSP** model

What is **MULTI-BSP**?

- A tree structure with nested components
- Where nodes have a storage capacity
- And leaves are processors
**What is MULTI-BSP?**

1. A tree structure with nested components
The MULTI-BSP model

What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
# The **MULTI-BSP** model

## What is **MULTI-BSP**?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
The MULTI-BSP model

What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors

MULTI-BSP

```
Multi_Core

core0
  th0  th1  th2  th3

core1
  th0  th1  th2  th3
```
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
The multi-BSP model

Execution model

A level $i$ superstep is:

- Level $i - 1$ executes code independently.
- Exchanges information with the $m_i$ memory.
- Synchronises with Level $i$.
A level $i$ superstep is:

- Level $i - 1$ executes code independently
Execution model

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges informations with the $m_i$ memory
The MULTI-BSP model

Execution model

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges informations with the $m_i$ memory
- Synchronises

\[ \text{Level } i \]
# Table of Contents

1. Introduction

2. Multi-ML
   - Overview
   - Primitives
   - Semantics
   - Typing
   - Implementation

3. Results

4. Conclusion
Basic ideas:

• bsml-like code on every stage of the multi-bsp architecture
• Specific syntax over ml: eases programming
• Multi-functions that recursively go through the tree.

V. Allombert et al.  BSPSP 2015
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
- *Multi-functions* that recursively go through the tree.
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
- *Multi-functions* that recursively go through the tree.

\[
\text{let } v = \langle \ldots \rangle
\]

\[
\langle \ldots \rangle = \langle e \rangle \langle e \rangle
\]
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] = where node = (* BSML code *)...
... ≪ f [args] ≫...
... in v
where leaf = (* OCAML code *)
... in v
```

V. Allombert et al.  BSPSP 2015  10 / 20
MULTI-ML: Tree recursion

Recursion structure

let multi f [args] =
  where node =
    (* BSML code *)
...
  ≪ f [args] ≫
  ... in v
where leaf =
  (* OCAML code *)
... in v
**MULTI-ML: Tree recursion**

Recursion structure

```ocaml
let multi f [args] =  
  where node =  
    (* BSML code *)  
    ...  
    f [args]  
    ... in v  
  where leaf =  
    (* OCAML code *)  
    ... in v
```

[Diagram of tree recursion]

- Result: `v`
**MULTI-ML: Tree recursion**

Recursion structure

```ml
let multi f [args] = 
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCAML code *)
    ... in v
```

Diagram: A tree structure with nodes labeled `N`, `f`, and leaves labeled `v`.
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
...
  ≪ f [args] ≫
  ... in v
where leaf =
  (* OCAML code *)
  ... in v
```

Result
**MULTI-ML: Tree recursion**

**Recursion structure**

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCAML code *)
    ... in v
```

![Tree recursion diagram](attachment:image.png)
**MULTI-ML: Tree recursion**

Recursion structure

```
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCAML code *)
    ... in v
```
**MULTI-ML: Tree recursion**

Recursion structure

```ocaml
let multi f [args] = 
  where node = 
    (* BSML code *)
    ...
    ⩾ f [args] ⩾
  ... in v

where leaf = 
  (* OCAML code *)
  ...
  ... in v
```

![Tree recursion diagram]

**Result**

\[ v_0.0 \]

\[ v_0.1 \]

\[ v_0.1.1 \]

\[ v_0.0.1 \]

\[ v_0.0.1.1 \]

\[ v_0.0.1.1.1 \]
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
  where node =
    ( (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    ( (* OCAML code *)
    ... in v
```

Diagram:

```
N ----------------- N
|                   |
|                   |
|                   |
|                   |
|                   |
v0.0.0 ------------ v0.0.1
|                 /
|                 |
|                 |
|                 |
|                 |
|                 |
v0.1.0 ------------ v0.1.1
```

Result

v0.0.0
v0.0.1
v0.1.0
v0.1.1
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    f [args]
  ...
 where leaf =
    (* OCAML code *)
  ...
```

![Tree diagram]
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
  ...
  ⪯ f [args] ⪰
  ... in v
  where leaf =
    (* OCAMLR code *)
  ... in v
```

Result

\[ \uparrow v_0 \]

Tree structure
**MULTI-ML:** Tree construction

Construction structure

```ocaml
let multi tree f [args] =
    where node =
        (* BSML code *)
        ... in
        (≪ f [args] ≫ , v)
    where leaf =
        (* OCAML code *)
        ... in v
```
**MULTI-ML: Tree construction**

**Construction structure**

```
let multi tree f [args] =
  where node =
    (⋆ BSML code ⋆)
    ... in
    (≪ f [args] ≫ , v)
  where leaf =
    (⋆ OCAML code ⋆)
    ... in v
```
**MULTI-ML: Tree construction**

**Construction structure**

```
let multi tree f [args] =
  where node =
    (* BSML code *)
    ... in
    (∇ f [args] ∇ , v)
  where leaf =
    (* OCAML code *)
    ... in v
```
MULTI-ML: Tree construction

Construction structure

\[
\text{let multi tree } f \ [\text{args}] = \\
\quad \text{where node } = \\
\quad \quad (\ast \text{ BSML code } \ast) \\
\quad \quad \ldots \text{ in } \\
\quad \quad (\ll f \ [\text{args}] \gg , v) \\
\quad \text{where leaf } = \\
\quad \quad (\ast \text{ OCAML code } \ast) \\
\quad \quad \ldots \text{ in } v
\]
MULTI-ML: Tree construction

Construction structure

```ocaml
let multi tree f [args] =
  where node =
    (* BSML code *)
    ... in
    (≪ f [args] ≫ , v)
  where leaf =
    (* OCAML code *)
    ... in v
```

![Tree diagram]

V. Allombert et al.

BSPSP 2015
**MULTI-ML: Tree construction**

**Construction structure**

```ocaml
let multi_tree f [args] =
  where node =
    (* BSML code *)
    ... in
    (∙ f [args] ∙ , v)
  where leaf =
    (* OCAML code *)
    ... in v
```

![Tree diagram](image)
MULTI-ML: Tree construction

Construction structure

```
let multi tree f [args] =
  where node =
    (* BSML code *)
    ... in
    (≪ f [args] ≫ , v)
  where leaf =
    (* OCAML code *)
    ... in v
```
MULTI-ML: Tree construction

Construction structure

```ocaml
define multi_tree f [args] =
  where node =
    (* BSML code *)
    ... in
    (f [args], v)
  where leaf =
    (* OCAML code *)
    ... in v
```

![Diagram of tree structure]
**MULTI-ML: Tree construction**

**Construction structure**

```ml
let multi tree f [args] =
  where node =
    (* BSML code *)
    ... in
    (≪ f [args] ≫, v)
  where leaf =
    (* OCAML code *)
    ... in v
```

![Tree Diagram]
MULTI-ML: Tree construction

Construction structure

```ocaml
let multi_tree f [args] =
  where node =
    (* BSML code *)
    ...
    in
    (≪ f [args] ≫ , v)
  where leaf =
    (* OCAML code *)
    ...
    in v
```

![Tree structure diagram]

V. Allombert et al.
BSPSP 2015
**MULTI-ML: Tree construction**

**Construction structure**

```ocaml
let multi tree f [args] =
  where node =
    (* BSML code *)
  ... in
  (≡ f [args] ≡, v)
where leaf =
  (* OCAML code *)
  ... in v
```
Primitives

Summary:
Primitives

Summary:
- $\text{special symbol}$

\begin{center}
\begin{tikzpicture}
  \node[circle,draw] {e} child {node[circle,draw] {e} child {node[draw] {e}} child {node[draw] {e}}}
  \node[circle,draw] {e} child {node[draw] {e}};
\end{tikzpicture}
\end{center}
Primitives

Summary:

- e
- gid

```
0.0
  0.0.0
  0.0.1
  0.1.0
  0.1.1
```

V. Allombert et al.
Summary:

- §e§
- gid
- at
Primitives

Summary:

- \$e\$
- \$gid\$
- \$at\$

V. Allombert et al.  
BSPSP 2015  
12 / 20
Primitives

Summary:

• $\$e\$
• $\text{gid}$
• $\text{at}$
• $\ll\ldots\text{f}\ldots\gg$
Summary:

- $\$e\$
- $\text{gid}$
- $\text{at}$
- $\langle \text{...f...} \rangle$
Primitives

Summary:

- §e§
- gid
- at
- ≪...f...≫
- #×#
Primitives

Summary:

- e
- gid
- at
- ...f...
- #x#
Primitives

Summary:

- $\$e\$
- gid
- at
- $\langle...f...\rangle$
- $#x#$

let $x = ...$

Diagram:

```
    x
   /\  \/
 #x# #x#
    /\  \/
   #x# #x#
```

V. Allombert et al.

BSPSP 2015
Primitives

Summary:

• $\_e\_e$
• $\text{gid}$
• $\text{at}$
• $\langle...f...\rangle$
• $#\times#$
• $\text{mkpar } f$

```
    mkpar (fun i → vi)
```

```
     /\            \
(./.)           ./ .
    /\            /\  \
  ./ .          ./ .
```

V. Allombert et al.  BSPSP 2015
Summary:

- §e§
- gid
- at
- ≪...f...≫
- #×#
- mkpar f

Primitives

```
mkpar (fun i → vi)
```

```
f 0; f 1
```
Primitives

Summary:

- §e§
- gid
- at
- ≪...f...≫
- #×#
- mkpar f

\[
\text{mkpar } (\text{fun } i \rightarrow v_i)
\]
Keep the intermediate results of the sum:

```ocaml
define multi_tree sum_list l =
  let node =
    let v = mkpar (fun i → split i l) in
    let rc = ≪ sum_list $v$ ≫ in
    let s = sumSeq (flatten ≪ at $rc$≫ ) in
    (rc,s)
  where leaf =
    sumSeq l
```

V. Allombert et al.  
BSPSP 2015  
13 / 20
Keep the intermediate results of the sum:

```ocaml
let multi_tree sum_list l =  
  where node =  
    let v = mkpar (fun i -> split i l) in  
    let rc = sum_list $v$ in  
    let s = sumSeq (flatten $rc$) in  
    (rc, s)  
  where leaf =  
    sumSeq l
```

[0...7]
Keep the intermediate results of the sum:

```ocaml
let multi_tree sum_list l =  
  where node =  
    let v = mkpar (fun i → split i l) in  
    let rc = ≪ sum_list $v$≫ in  
    let s = sumSeq (flatten ≪ at $rc$≫ ) in  
    (rc,s)  
  where leaf =  
    sumSeq l
```
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let rc = sum_list $v$ in
    let s = sumSeq (flatten $rc$) in
    (rc,s)
  where leaf =
    sumSeq l
```
Keep the intermediate results of the sum:

Let multi tree sum_list l =

where node =

let v = mkpar (fun i → split i l) in
let rc = ≪ sum_list $v$≫ in
let s = sumSeq (flatten ≪ at $rc$≫ ) in (rc,s)

where leaf =

sumSeq l
Keep the intermediate results of the sum:

```ocaml
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = \[ sum_list $v$ \] in
    let s = sumSeq (flatten \[ at $rc$ \]) in
    (rc, s)
  where leaf =
    sumSeq l
```

V. Allombert et al.

BSPSP 2015
Keep the intermediate results of the sum:

```ocaml
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let rc = ≪ sum_list $v$≫ in
    let s = sumSeq (flatten ≪ at $rc$≫ ) in
    (rc,s)
  where leaf =
    sumSeq l
```

V. Allombert et al.  
BSPSP 2015  
13 / 20
Keep the intermediate results of the sum:

```ml
let multi_tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let rc = ≪ sum_list $v$≫ in
    let s = sumSeq (flatten ≪ at $rc$≫ ) in
    (rc, s)
  where leaf =
    sumSeq l
```

```text
Keep the intermediate results of the sum:

let multi_tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let rc = ≪ sum_list $v$≫ in
    let s = sumSeq (flatten ≪ at $rc$≫ ) in
    (rc, s)
  where leaf =
    sumSeq l
```
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
  let v = mkpar (fun i → split i l) in
  let rc = ≪ sum_list $v$≫ in
  let s = sumSeq (flatten ≪ at $rc$≫ ) in
  (rc,s)
  where leaf =
    sumSeq l
```
Semantics

Formal definition of a core-language

Useful for:
- Study of properties
- Proof of programs/compiler/typing rules

Currently
- Inductive big-step: confluent
- Co-inductive: mutually exclusive
### Typing

**Purely Constraint-Based system**: PCB(X)

- Constraint based
- Extension of DM’s type system
- Easy to extend
- Related to HM(X)

---

**MULTI-ML type extension**

- Add parallel constructions
- Introduce locality \((s, \ell, b \text{ and } m)\) using effects
- Reject nested vectors
- Consistency
Implementation

Sequential simulator

- OCAML-like toplevel
- Test and debug
- Tree structure
- Hash tables to represent memories

```ocaml
#let multi tree f n =
  where node =
    let r = f ($pid$ + #n# + 1) in
    (r, (gid^^"="^^"^n))
  where leaf =
    (gid^^"="^^"^n);

let f : int -> string tree = <multi-fun>
#(f 0)
o "0 -> 0"
  o "0.0 -> 1"
    o "0.0.0 -> 2"
    o "0.0.1 -> 3"
  o "0.1 -> 2"
    o "0.1.0 -> 3"
    o "0.1.1 -> 4"
```
## Distributed implementation

### Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on mpi
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
- Shared/Distributed memory optimisations
## Distributed implementation

### Our approach

- Modular

- Generic functors

- Communication routines

- Portable on shared and distributed memories

### Current version

- Based on mpi
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
- Shared/Distributed memory optimisations
### Distributed implementation

**Our approach**

- Modular
- Generic functors
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
Distributed implementation

Our approach

• Modular
• Generic functors
• Communication routines
• Portable on shared and distributed memories

Current version

• Based on MPI
• SPMD
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
- SPMD
- One process for each nodes/leaves
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
- Shared/Distributed memory optimisations
Table of Contents

1 Introduction

2 Multi-ML

3 Results

4 Conclusion
Naive Eratosthenes algorithm

- $\sqrt{n}$th first prime numbers
- Based on scan
- Unbalanced
Naive Eratosthenes algorithm

- $\sqrt{n}$th first prime numbers
- Based on scan
- Unbalanced

Benchmarks

Mirev 3

0 ... 7 ... ... ... ... ... ... ... 56 ... 63
### Naive Eratosthenes algorithm

- \(\sqrt(n)\)th first prime numbers
- Based on scan
- Unbalanced

### Results

<table>
<thead>
<tr>
<th></th>
<th>100_000</th>
<th></th>
<th>500_000</th>
<th></th>
<th>1_000_000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MULTI-ML</td>
<td>BSML</td>
<td>MULTI-ML</td>
<td>BSML</td>
<td>MULTI-ML</td>
</tr>
<tr>
<td>1</td>
<td>0.7</td>
<td>1.8</td>
<td>22.4</td>
<td>105.0</td>
<td>125.3</td>
</tr>
<tr>
<td>16</td>
<td>0.5</td>
<td>0.45</td>
<td>2.1</td>
<td>5.2</td>
<td>4.7</td>
</tr>
<tr>
<td>64</td>
<td>0.3</td>
<td>0.3</td>
<td>1.3</td>
<td>8.7</td>
<td>4.1</td>
</tr>
<tr>
<td>256</td>
<td>0.2</td>
<td>0.29</td>
<td>1.1</td>
<td>7.4</td>
<td>3.8</td>
</tr>
<tr>
<td>128</td>
<td>0.3</td>
<td>0.3</td>
<td>1.3</td>
<td>8.7</td>
<td>4.1</td>
</tr>
</tbody>
</table>
Table of Contents

1. Introduction
2. Multi-ML
3. Results
4. Conclusion
Conclusion

MULTI-ML
Conclusion

MULTI-ML

• Recursive multi-functions
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
**Conclusion**

### MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
**MULTI-ML**

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work

- Optimise mpi implementation
- Type system for multi-ml
- Real life benchmarks

V. Allombert et al.  
BSPSP 2015
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confluent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work

- Optimise MPI implementation
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work

- Optimise MPI implementation
- Type system for MULTI-ML
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work

- Optimise MPI implementation
- Type system for MULTI-ML
- Real life benchmarks

V. Allombert et al.

BSPSP 2015
Any questions?