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

UNIVERSITÉ
PARIS-EST CRÉTEIL
VAL DE MARNE

Table of Contents

Table of Contents

Ocaml : a ML language

Strengths of Ocaml

- A functionnal programming language

Ocaml : a ML language

OCaml

Strengths of Ocaml

- A functionnal programming language
- A powerful type system

Ocaml : a ML language

OCaml

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

OCaml

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

```
# 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
```

What is BSML?

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?

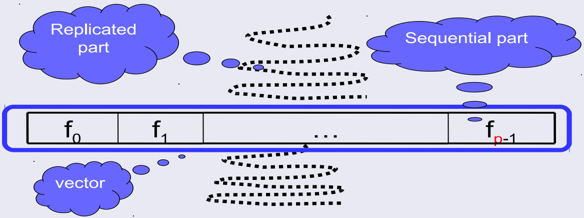
- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
- Formal sementics $\rightarrow$ computer-assisted proofs (COQ)

What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
- Formal sementics → computer-assisted proofs (COQ)

Main idea

Parallel data structure ⇒ vectors:



Asynchronous primitives

Asynchronous primitives

- $\ll e \gg \; : \langle e, \ldots, e \rangle$

Asynchronous primitives

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

Asynchronous primitives

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

BSML primitives

Asynchronous primitives

- $\ll$ e $\gg$: $\langle \mathsf{e}, \ldots, \mathsf{e} \rangle$
- \$v\$: v_i on processor i, assumes $\mathsf{v} \equiv \langle \mathsf{v}_0, \ldots, \mathsf{v}_{\mathbf{p}-1} \rangle$
- \$**pid**\$: A predefined vector: i on processor i

Synchronous primitives

- **proj** : $\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto (\textbf{fun } i \rightarrow x_i)$

Asynchronous primitives

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

Synchronous primitives

- **proj** : $\langle x_0, \ldots, x_{\mathbf{p}-1}\rangle \mapsto (\mathbf{fun}\ i \rightarrow x_i)$
- **put** : $\langle f_0, \ldots, f_{\mathbf{p}-1}\rangle \mapsto \langle(\mathbf{fun}\ i \rightarrow f_i\ 0), \ldots, (\mathbf{fun}\ i \rightarrow f_i\ (\mathbf{p}-1))\rangle$

Code example

For a BSP machine with 3 processors:

```
# let vec = ≪ "Hello␣" ≫ ;;
val vec : string par = <"Hello␣", "Hello␣", "Hello␣">
# let vec2 = ≪ $vec$^(string_of_int $pid$) ≫ ;;
val vec2 : string par = <"Hello␣0", "Hello␣1", "Hello␣2">
# let totex v = List.map (proj v) procs;;
val totex : 'a Bsml.par → 'a list = <fun>
# totex vec2;;
— : string list = ["Hello0"; "Hello1"; "Hello2"]
```

What is MULTI-BSP?

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

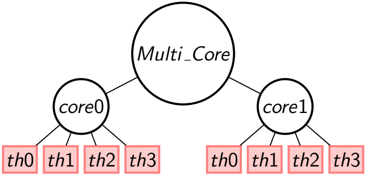
What is MULTI-BSP?

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

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

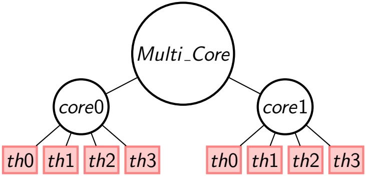


The MULTI-BSP model

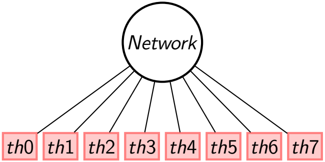
What is MULTI-BSP?

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

MULTI-BSP

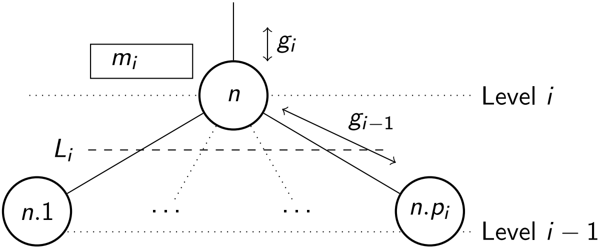


BSP



Execution model

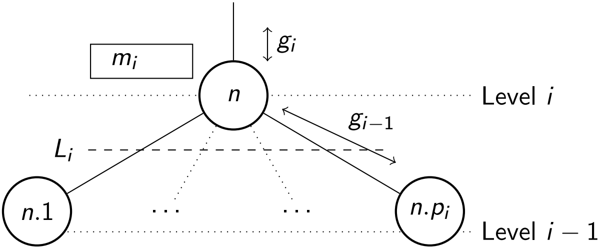
A level i superstep is:



Execution model

A level i superstep is:

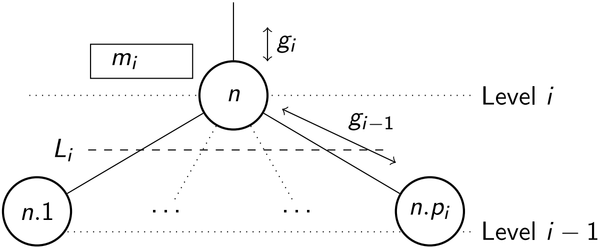
- Level $i-1$ executes code independantly



Execution model

A level i superstep is:

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



Execution model

A level i superstep is:

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

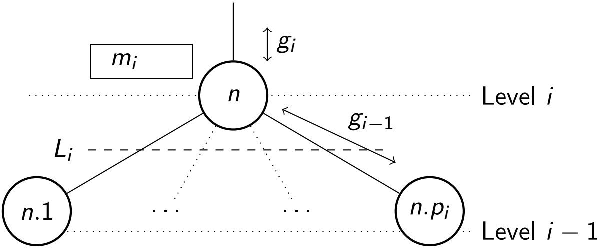


Table of Contents

Basic ideas:

Basic ideas:

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

Basic ideas:

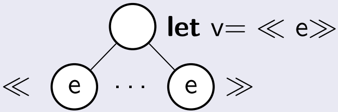
- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming

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.



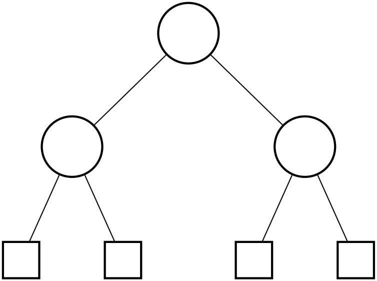
Recursion structure

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

MULTI-ML: Tree recursion

Recursion structure

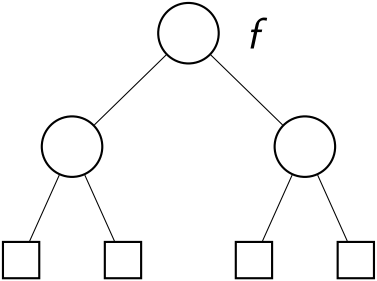
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

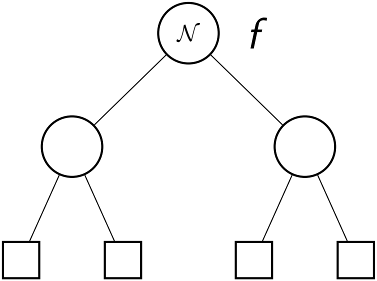
Recursion structure

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



Recursion structure

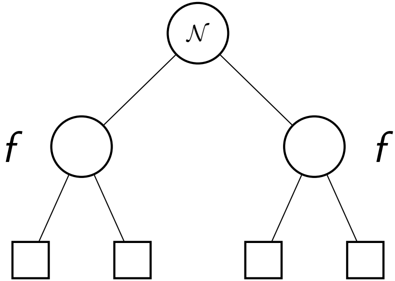
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

Recursion structure

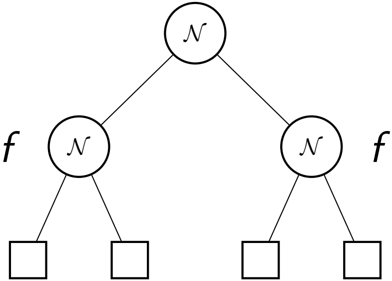
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

Recursion structure

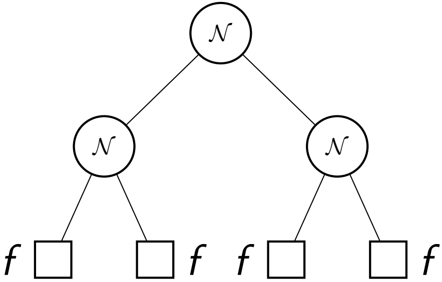
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

Recursion structure

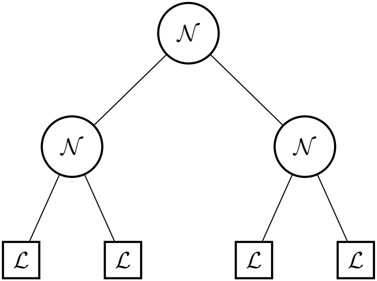
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

Recursion structure

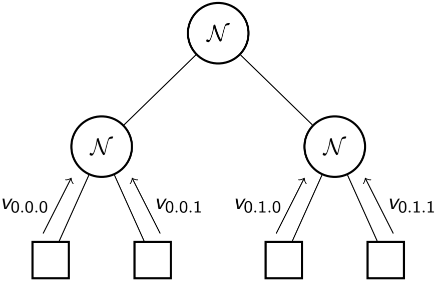
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

Recursion structure

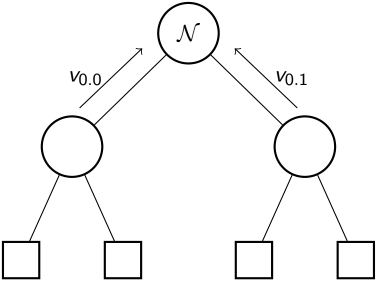
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ... **in** v
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree recursion

Recursion structure

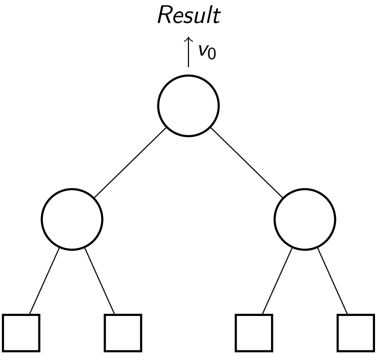
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... **in** v
  where leaf =
    (* OCAML code *)
    ... **in** v



MULTI-ML: Tree recursion

Recursion structure

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



MULTI-ML: Tree construction

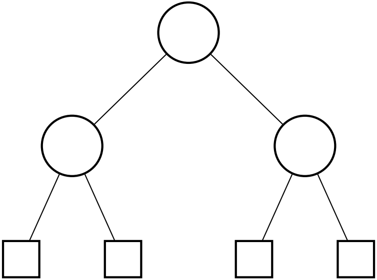
Construction structure

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

MULTI-ML: Tree construction

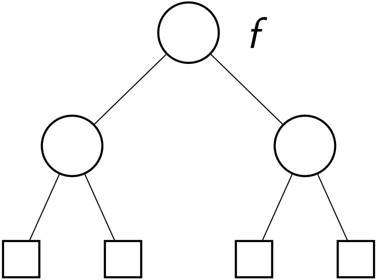
Construction structure

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



Construction structure

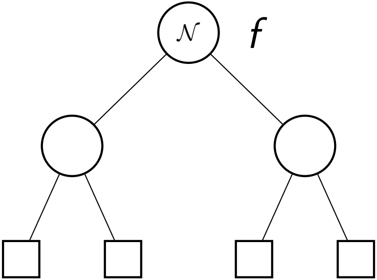
let multi tree f [args] =
  where node =
   (* BSML code *)
   ... **in**
   (≪ f [args] ≫ , v)
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree construction

Construction structure

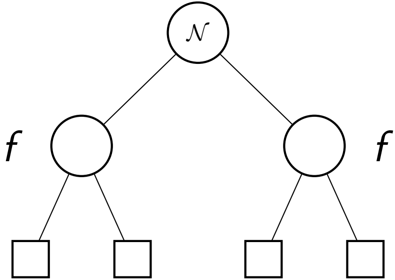
let multi tree f [args] =
  where node =
   (∗ BSML code ∗*)*
   … **in**
   (≪ f [args] ≫ , v)
  where leaf =
   (∗ OCAML code ∗*)*
   … **in** v



MULTI-ML: Tree construction

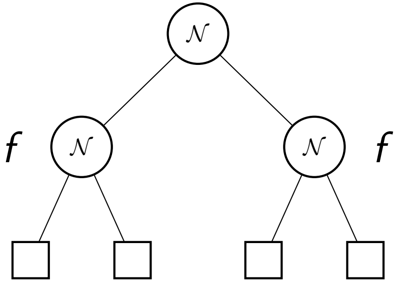
Construction structure

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



Construction structure

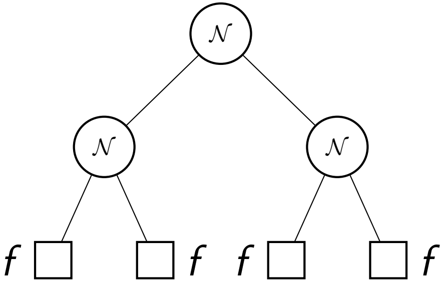
let multi tree f [args] =
  where node =
   (* BSML code *)
   ... **in**
   (≪ f [args] ≫ , v)
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree construction

Construction structure

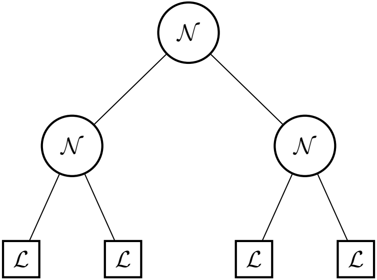
let multi tree f [args] =
  where node =
   (BSML code *)*
   ... **in**
   (≪ f [args] ≫ , v)
  where leaf =
   (OCAML code *)*
   ... **in** v



MULTI-ML: Tree construction

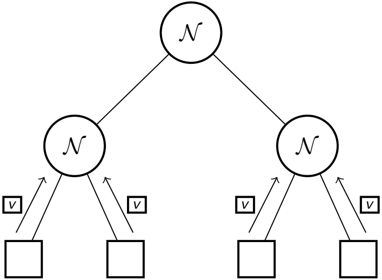
Construction structure

let multi tree f [args] =
  where node =
   (* BSML code *)
   ... **in**
   ($\ll$ f [args] $\gg$, v)
  where leaf =
   (* OCAML code *)
   ... **in** v



Construction structure

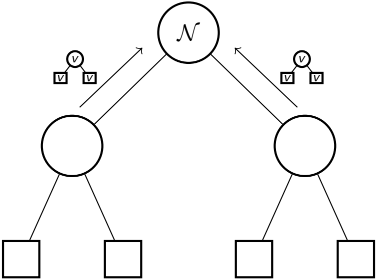
let multi tree f [args] =
  where node =
   (* BSML code *)
   ... **in**
   (≪ f [args] ≫ , v)
  where leaf =
   (* OCAML code *)
   ... **in** v



MULTI-ML: Tree construction

Construction structure

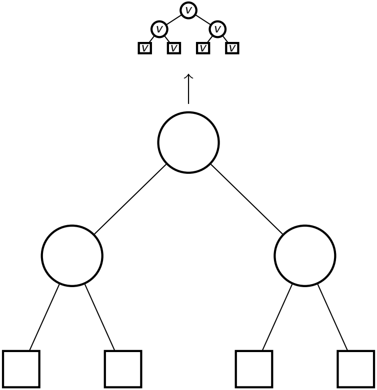
let multi tree f [args] =
 where node =
  (* BSML code *)
  ... **in**
  (≪ f [args] ≫ , v)
 where leaf =
  (* OCAML code *)
  ... **in** v



MULTI-ML: Tree construction

Construction structure

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



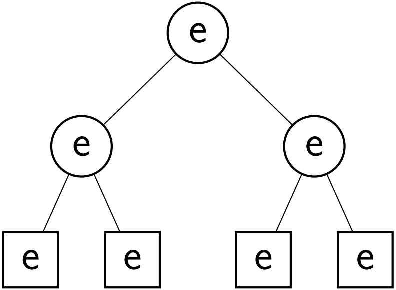
Primitives

Summary:

Primitives

Summary:

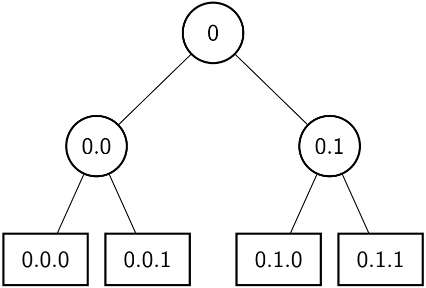
- §e§



Primitives

Summary:

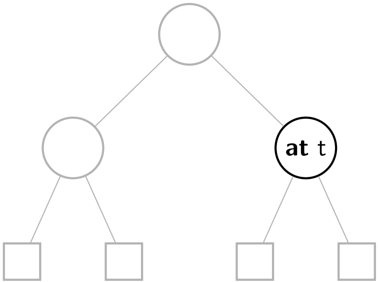
- §e§
- **gid**



Primitives

Summary:

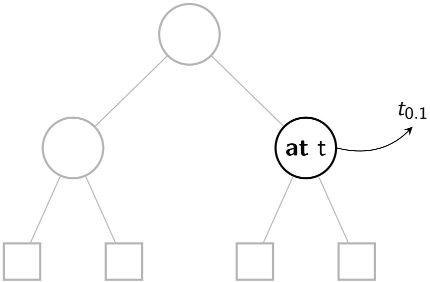
- §e§
- **gid**
- **at**



Primitives

Summary:

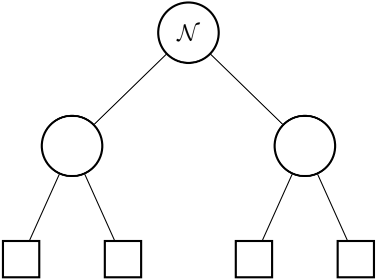
- §e§
- **gid**
- **at**



Primitives

Summary:

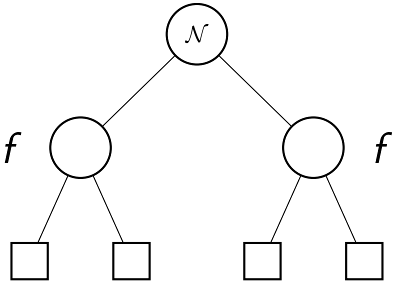
- §e§
- **gid**
- **at**
- ≪...f...≫



Primitives

Summary:

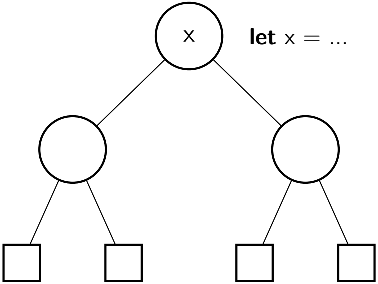
- §e§
- **gid**
- **at**
- ≪...f...≫



Primitives

Summary:

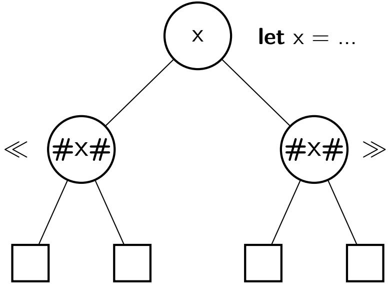
- §e§
- **gid**
- **at**
- ≪...f...≫
- **#**x**#**



Primitives

Summary:

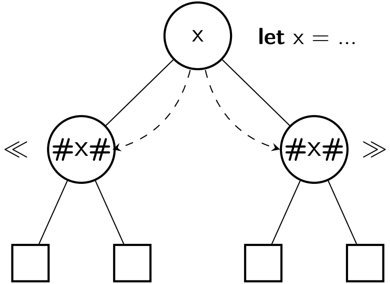
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#



Primitives

Summary:

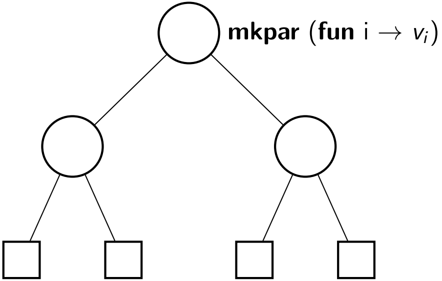
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#



Primitives

Summary:

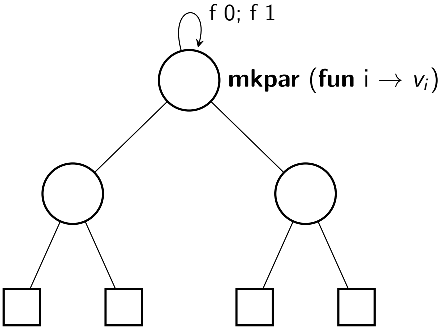
- §e§
- **gid**
- **at**
- ≪...f...≫
- #×#
- **mkpar** f



Primitives

Summary:

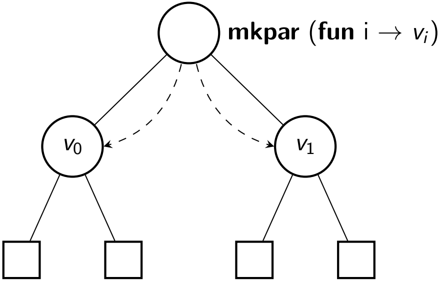
- §e§
- **gid**
- **at**
- ≪...f...≫
- **#**×**#**
- **mkpar** f



Primitives

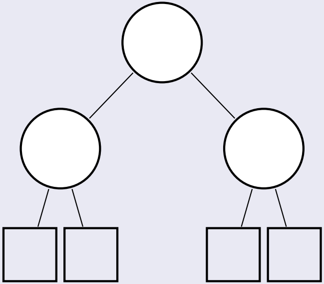
Summary:

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



Code example

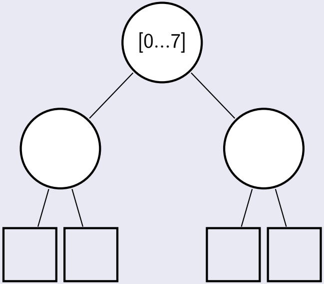
Keep the intermediate results of the sum:



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

Code example

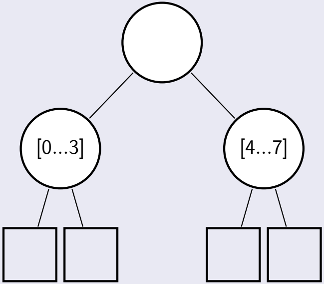
Keep the intermediate results of the sum:



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

Code example

Keep the intermediate results of the sum:

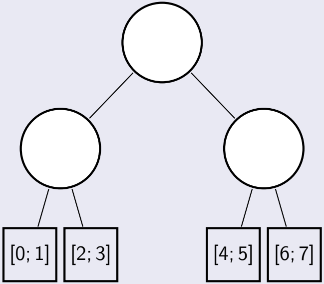


let multi tree sum_list l =
  where node =
    let v = **mkpar** (**fun** i → split i l) **in**
    let rc = ≪ sum_list \$v\$≫ **in**
    let s = sumSeq (flatten ≪ **at** \$rc\$≫) **in**
    (rc,s)
  where leaf =

    sumSeq l

Code example

Keep the intermediate results of the sum:

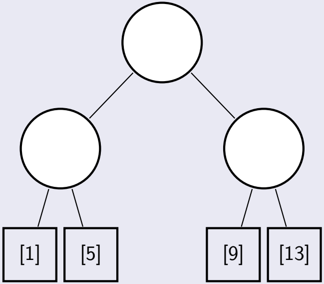


let multi tree sum_list l =
  where node =
    let v = **mkpar** (**fun** i → split i l) **in**
    let rc = ≪ sum_list \$v\$≫ **in**
    let s = sumSeq (flatten ≪ **at** \$rc\$≫) **in**
    (rc,s)
  where leaf =

    sumSeq l

Code example

Keep the intermediate results of the sum:

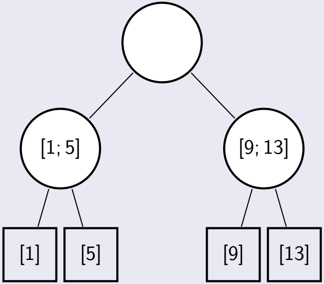



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i → split i l) in
      let rc = ≪ sum_list $v$≫ in
      let s = sumSeq (flatten ≪ at $rc$≫ ) in
      (rc,s)
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum:

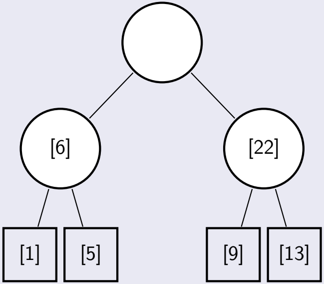



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i → split i l) in
      let rc = ≪ sum_list $v$≫ in
      let s = sumSeq (flatten ≪ at $rc$≫ ) in
      (rc,s)
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum:

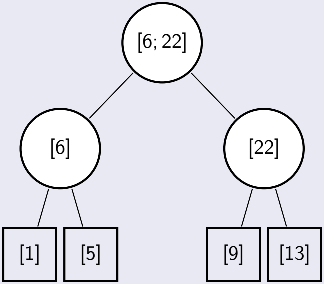



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i → split i l) in
      let rc = ≪ sum_list $v$≫ in
      let s = sumSeq (flatten ≪ at $rc$≫ ) in
      (rc,s)
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum:

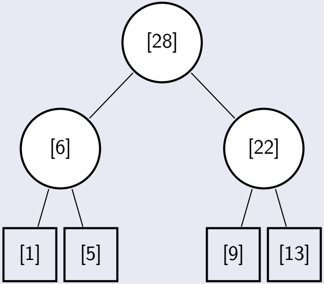


let multi tree sum_list l =
  where node =
    let v = **mkpar** (**fun** i → split i l) **in**
    let rc = ≪ sum_list \$v\$≫ **in**
    let s = sumSeq (flatten ≪ **at** \$rc\$≫) **in**
    (rc,s)
  where leaf =

    sumSeq l

Code example

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-languge

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, ℓ, b and m) using effects
- Reject nested vectors
- Concistency

Implementation

Sequential simulator

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

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

— : val f : int→string tree = <multi-fun>
#(f 0)
o "0→ 0"
|
−−o "0.0→ 1"
| | −→ "0.0.0→ 2"
| | −→ "0.0.1→ 3"
−−o "0.1→ 2"
| | −→ "0.1.0→ 3"
| | −→ "0.1.1→ 4"
```

Distributed implementation

Our approach

Distributed implementation

Our approach

- Modular

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

Benchmarks

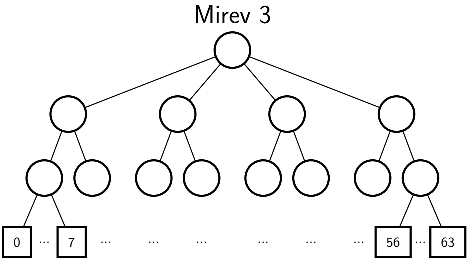
Naive Eratosthenes algorithm

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

Benchmarks

Naive Eratosthenes algorithm

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

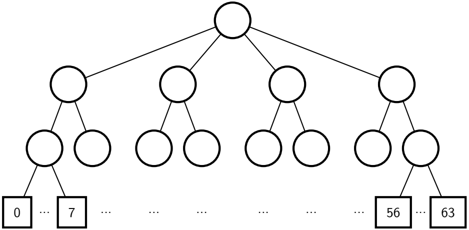


Benchmarks

Naive Eratosthenes algorithm

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

Mirev 3



Results

	100_000		500_000		1_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	22.4	105.0	125.3	430.7
64	0.3	0.3	1.3	8.7	4.1	56.1
128	0.5	0.45	2.1	5.2	4.7	24.3

Table of Contents

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)

Conclusion

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

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

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

Merci !

Any questions ?