Programming MULTI-BSP Algorithms in ML

Victor ALLOMBERT

LIFO - Université d'Orléans

15 January 2018

Table of Contents

1. Introduction
2. The MULTI-ML language
3. Type system
4. Implementation
5. Conclusion

Table of Contents

1. Introduction
   - The world of parallel computing
2. The MULTI-ML language
3. Type system
4. Implementation
5. Conclusion

The world of parallel computing

Simulations:

Fluid simulation
3D Visualisation

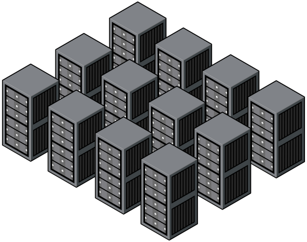
Big-Data:

IoT
Social Networking
Data science

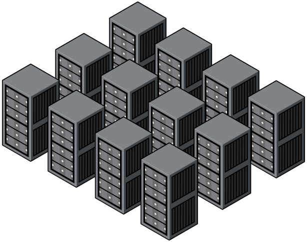
Symbolic computation:

Model-Checking
Formal computing

Super-computer

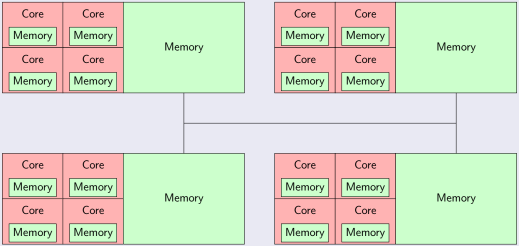


Hierarchical architectures

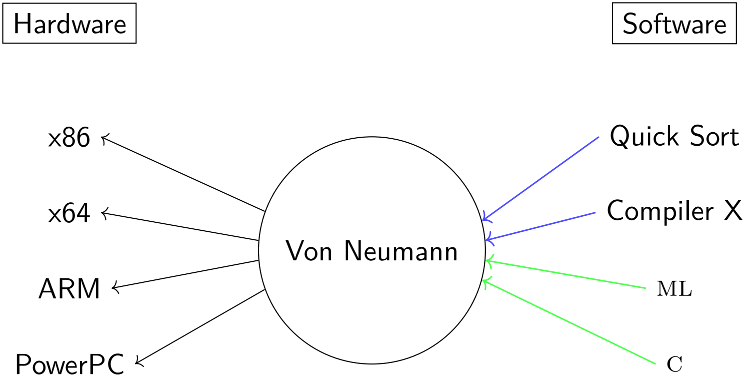


Characterised by:

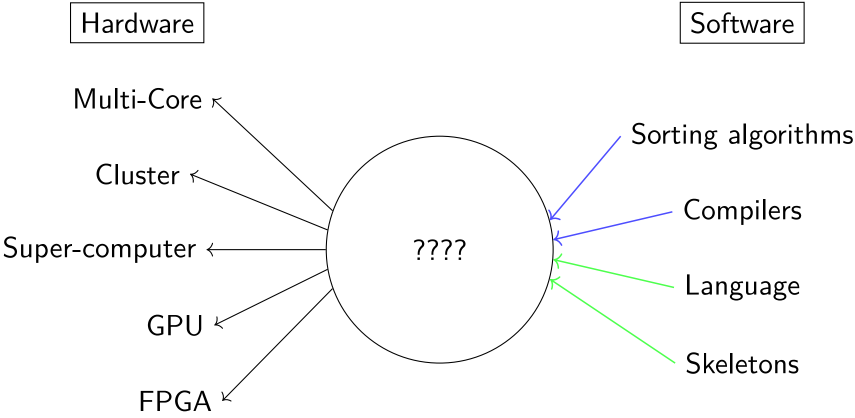
- Interconnected units
- Both shared and distributed memories
- Hierarchical memories



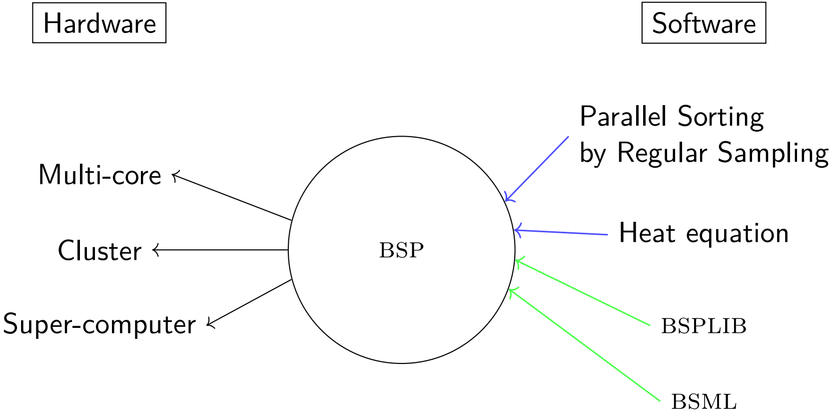
A sequential bridging model



A parallel bridging model



A parallel bridging model



Bulk Synchronous Parallelism

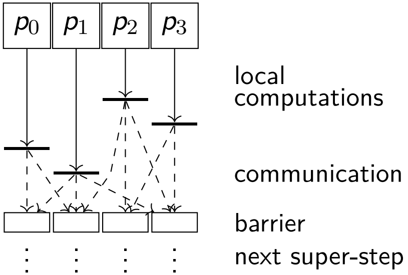
The BSP computer

Defined by:

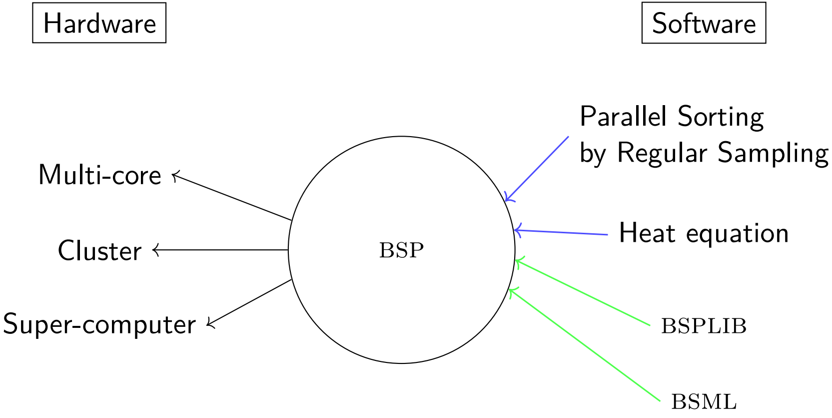
- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- Deadlock-free
- Predictable performances

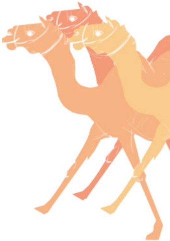


A parallel bridging model



What is BSML?

- Explicit BSP programming with a functional approach



What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML and implemented over OCAML

What is BSML?

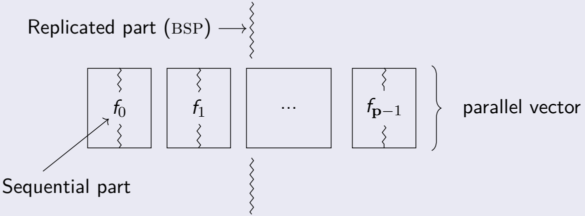
- Explicit BSP programming with a functional approach
- Based upon ML and implemented over OCAML
- Formal semantics $\rightarrow$ computer-assisted proofs (COQ)

What is BSML?

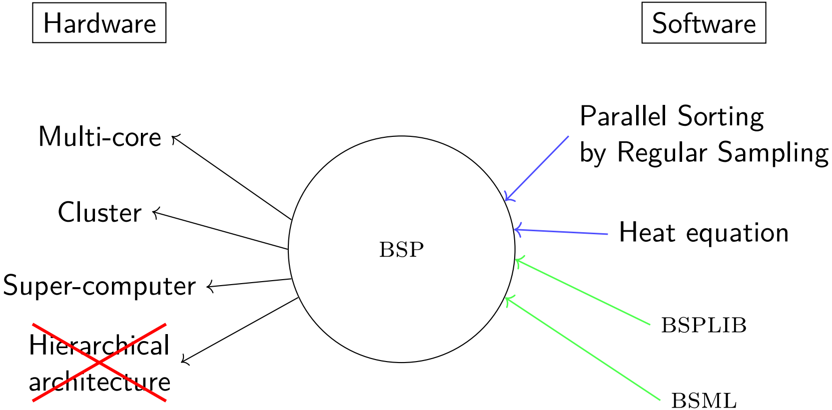
- Explicit BSP programming with a functional approach
- Based upon ML and implemented over OCAML
- Formal semantics → computer-assisted proofs (COQ)

Main idea

Parallel data structure ⇒ *parallel vector*:



A parallel bridging model



A parallel bridging model

Hardware

Multi-core

Cluster

Super-comp

~~Hierarchic architecture~~

Software

Parallel Sorting
by Regular Sampling

eat equation

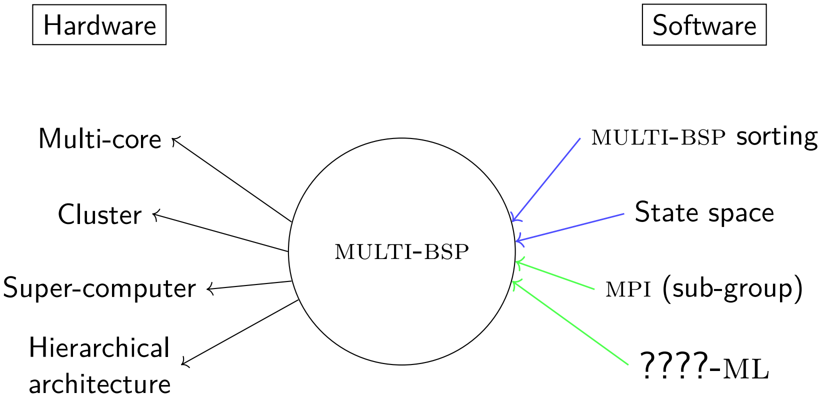
BSPLIB

BSML

Why ?

- Flat memories
- No sub-synchronisation

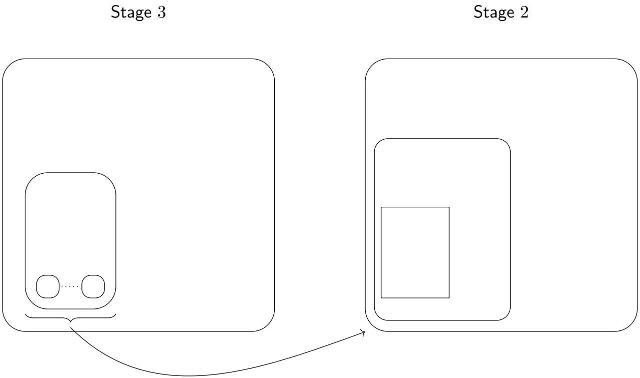
A parallel bridging model



What is MULTI-BSP?

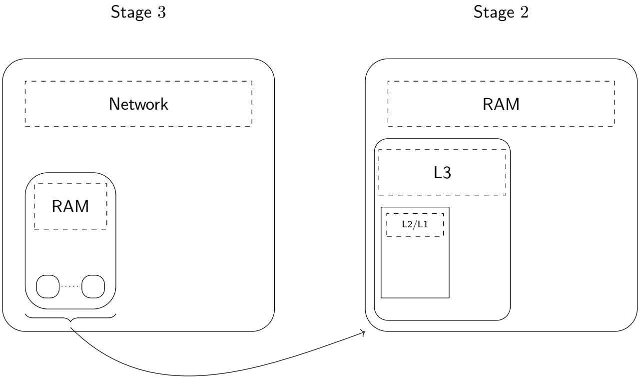
What is MULTI-BSP?

1. A tree structure with nested components



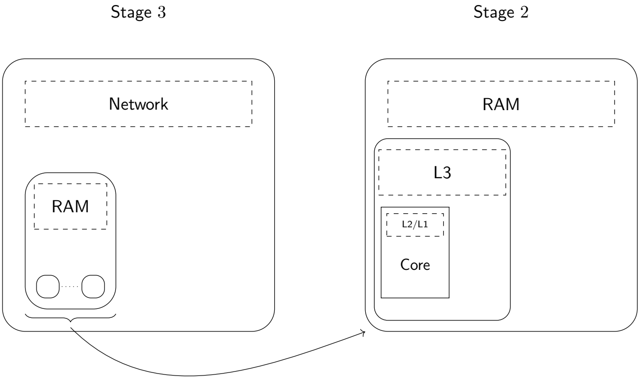
What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity



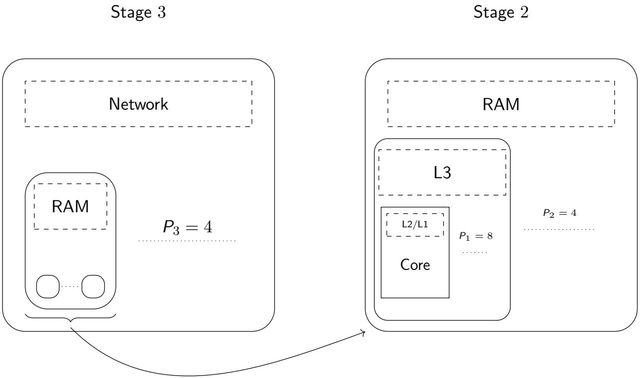
What is MULTI-BSP?

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



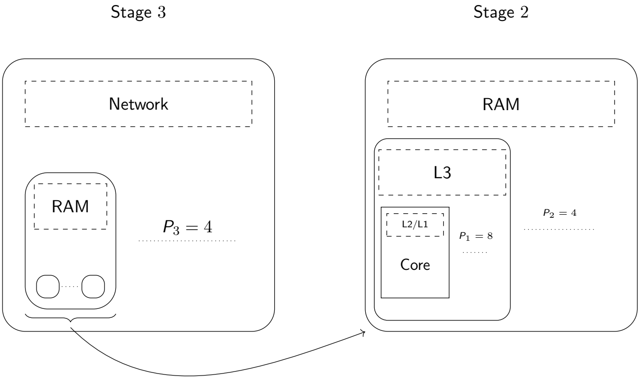
What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
4. With sub-synchronisation capabilities



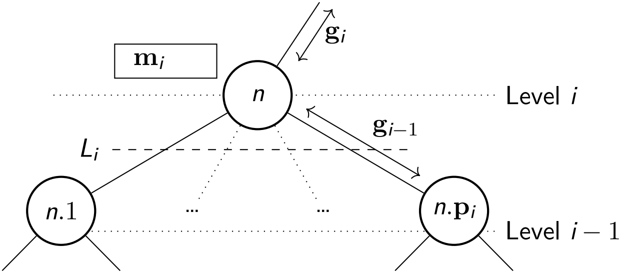
What is MULTI-BSP?

- Stage 3: 4 nodes with a network access
- Stage 2: one node has 4 chips plus RAM
- Stage 1: one chip has 8 cores plus L3 cache
- Stage 0: one core with L1/L2 caches



Execution model

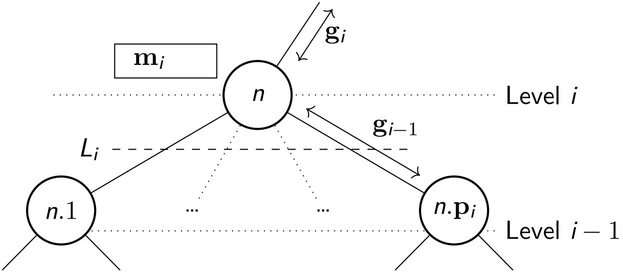
A level i superstep is:



Execution model

A level i superstep is:

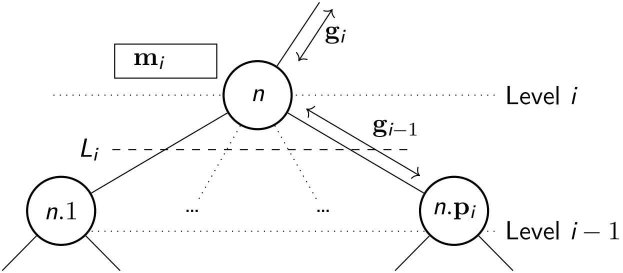
- Level $i-1$ executes code independently



Execution model

A level i superstep is:

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



Execution model

A level i superstep is:

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

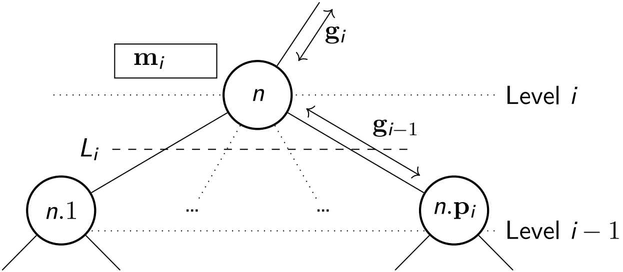


Table of Contents

1 Introduction

2 The MULTI-ML language
- MULTI-ML overview
- The MULTI-ML primitives
- A code example

3 Type system

4 Implementation

5 Conclusion

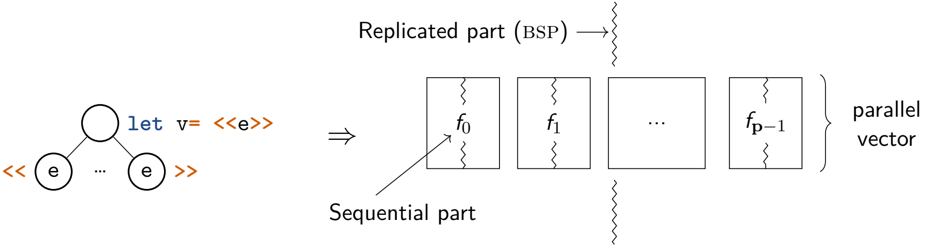
The MULTI-ML language

Basic ideas

The MULTI-ML language

Basic ideas

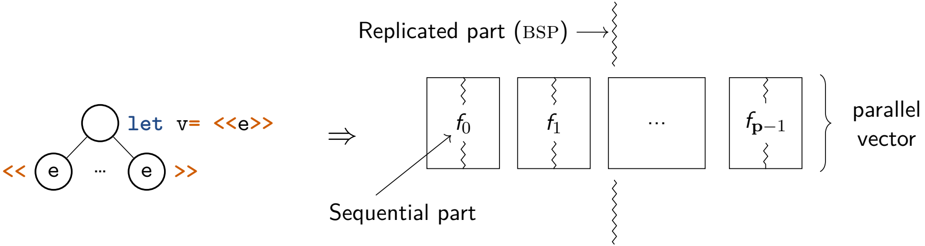
- BSML-like code on every stage of the MULTI-BSP architecture



The MULTI-ML language

Basic ideas

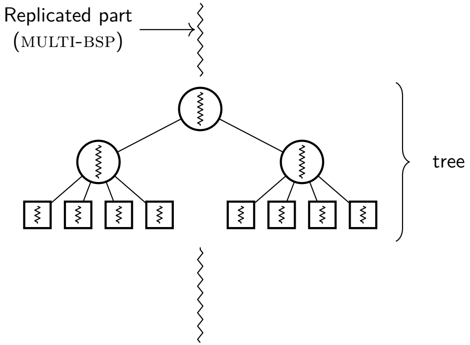
- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming



The MULTI-ML language

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 MULTI-BSP tree



MULTI-ML: Tree recursion

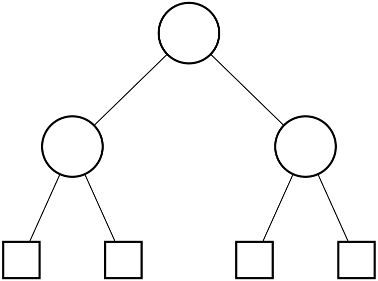
Recursion structure

```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```

MULTI-ML: Tree recursion

Recursion structure

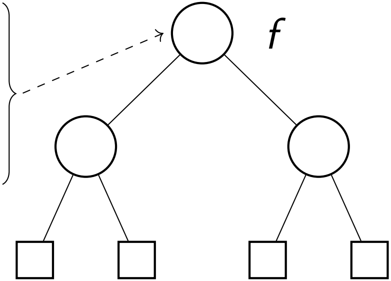
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

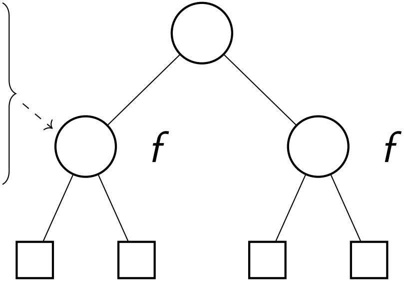
Recursion structure

```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



Recursion structure

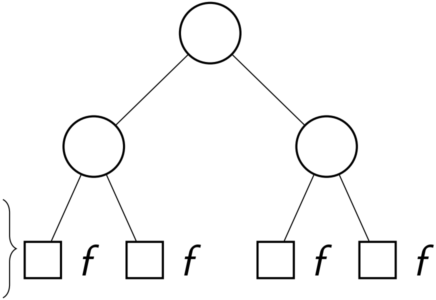
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

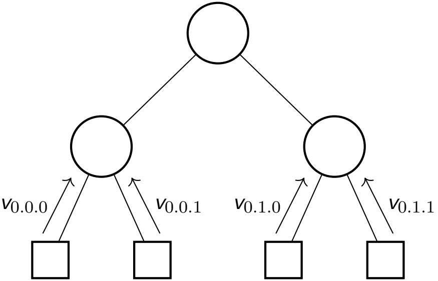
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

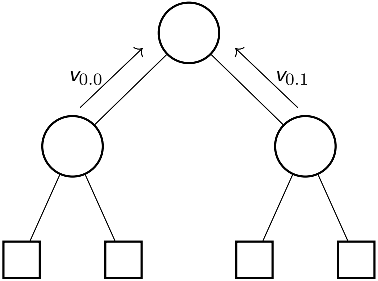
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

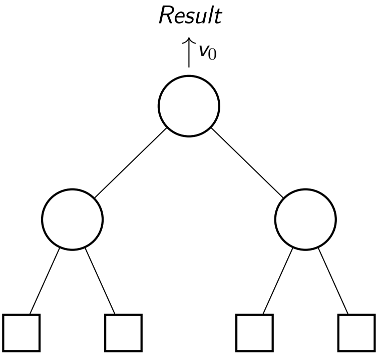
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



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

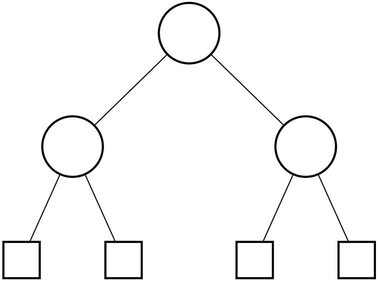
Tree construction

```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```

MULTI-ML: Tree construction

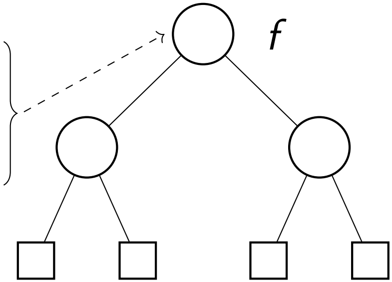
Tree construction

```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



Tree construction

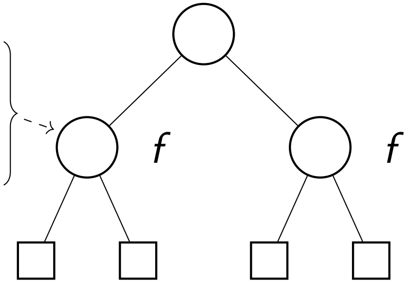
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

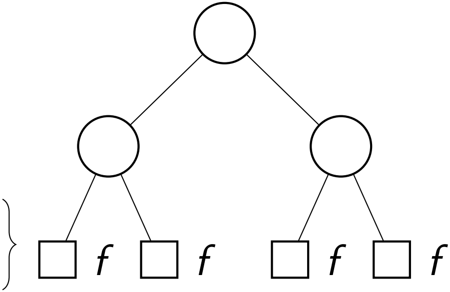
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

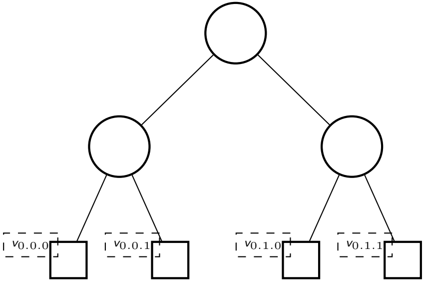
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

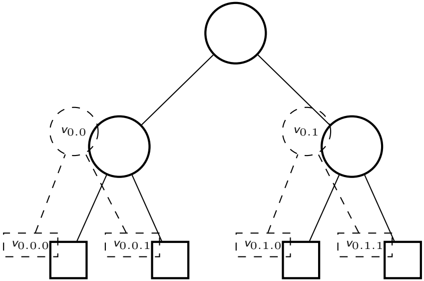
Tree construction

```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



Tree construction

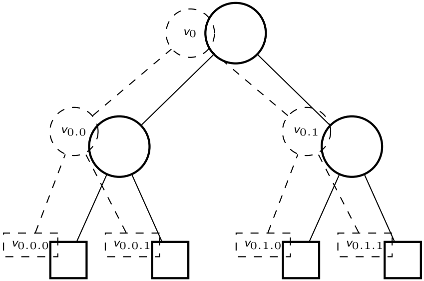
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



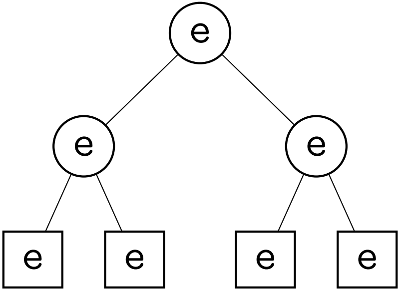
Primitives

Summary

Primitives

Summary

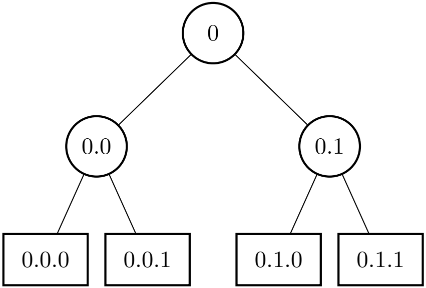
- `mktree` e



Primitives

Summary

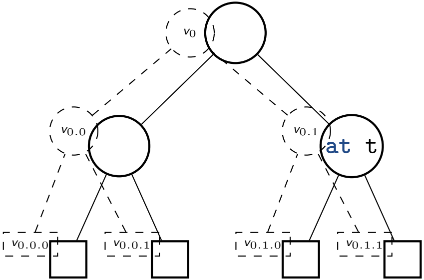
- `mktree e`
- `gid`



Primitives

Summary

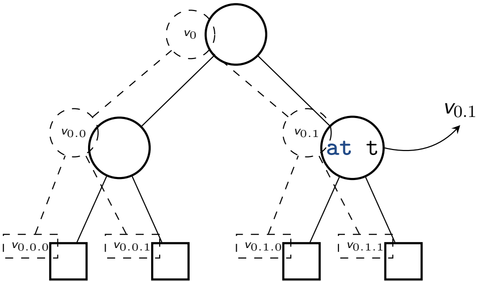
- `mktree e`
- `gid`
- `at`



Primitives

Summary

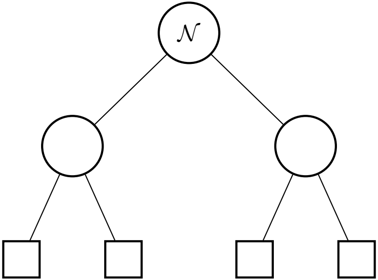
- `mktree e`
- `gid`
- `at`



Primitives

Summary

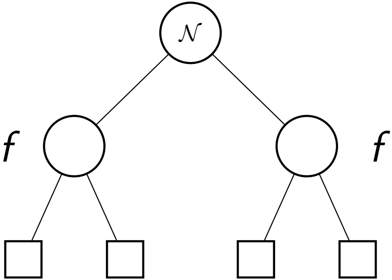
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`



Primitives

Summary

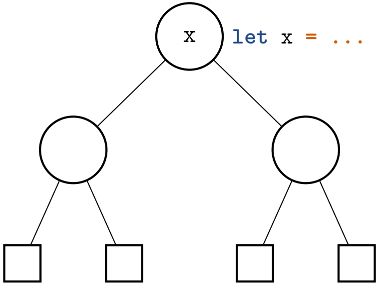
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`



Primitives

Summary

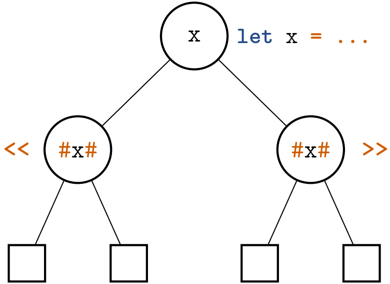
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

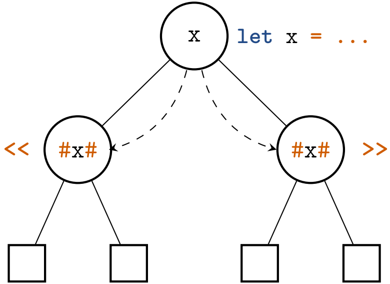
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

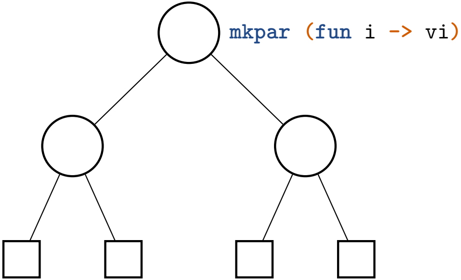
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

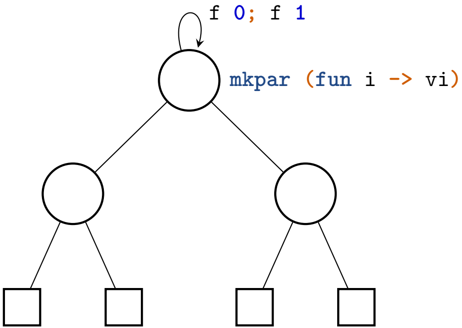
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Primitives

Summary

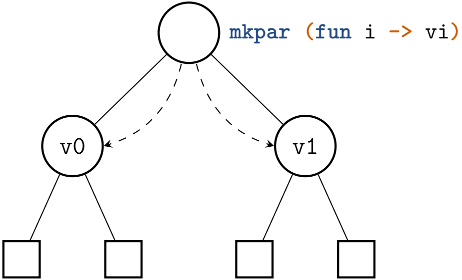
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Primitives

Summary

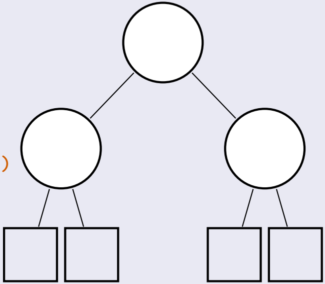
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Code example

Keep the intermediate results of the sum

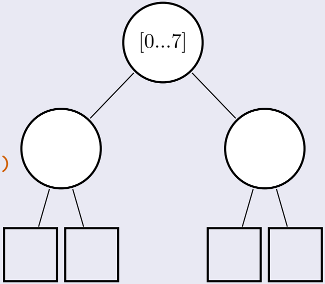
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =
        sumSeq l
```



Code example

Keep the intermediate results of the sum

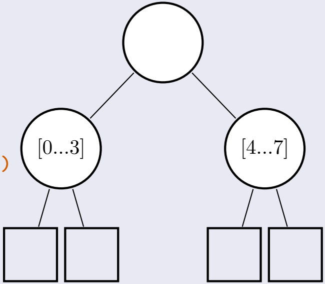
```
let multi tree sum_list l =
    where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
    where leaf =

      sumSeq l
```



Code example

Keep the intermediate results of the sum

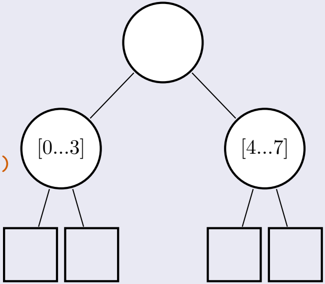
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =

        sumSeq l
```



Code example

Keep the intermediate results of the sum

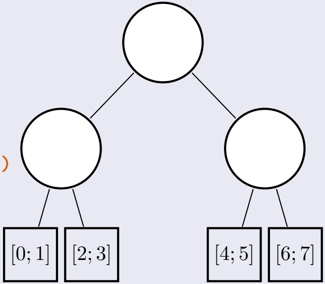
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =
        sumSeq l
```



Code example

Keep the intermediate results of the sum

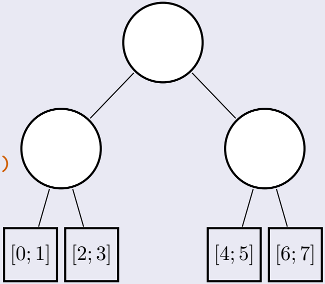
```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in finally rc s
  where leaf =

    sumSeq l
```



Code example

Keep the intermediate results of the sum

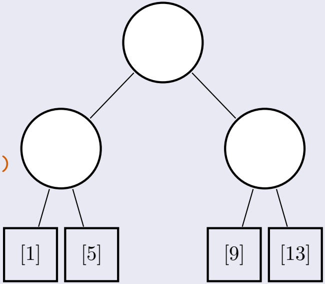
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =
        sumSeq l
```



Code example

Keep the intermediate results of the sum

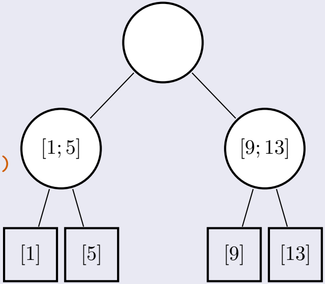
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =
        sumSeq l
```



Code example

Keep the intermediate results of the sum

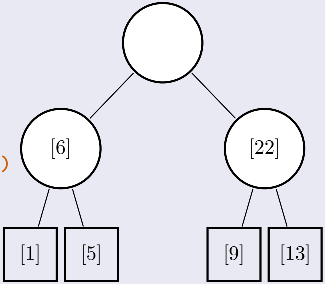
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =

        sumSeq l
```



Code example

Keep the intermediate results of the sum

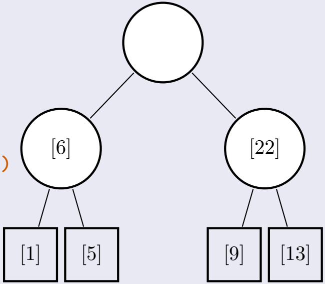
```
let multi tree sum_list l =
    where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
    where leaf =

      sumSeq l
```



Code example

Keep the intermediate results of the sum

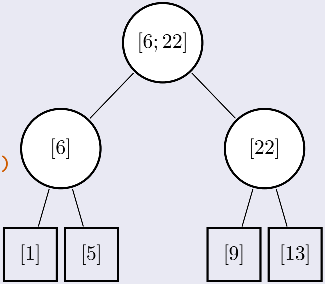
```
let multi tree sum_list l =
    where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
    where leaf =
      sumSeq l
```



Code example

Keep the intermediate results of the sum

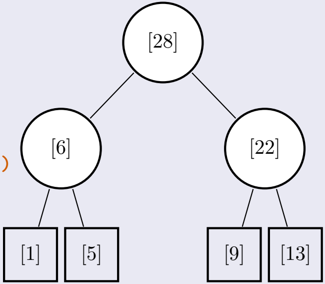
```
let multi tree sum_list l =
     where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
        in finally rc s
     where leaf =
        sumSeq l
```



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 finally 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 finally rc s
     where leaf =

        sumSeq l
```

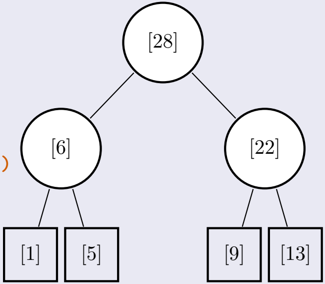


Table of Contents

1. Introduction
2. The MULTI-ML language
3. Type system
   - Parallel program safety
   - The MULTI-ML typing system
4. Implementation
5. Conclusion

Parallel program safety

- Replicated coherency

Replicated coherency

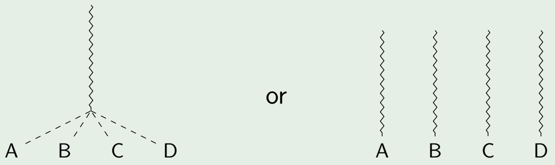
```
if random_bool () then
  multi_fct ()
else
   (fun _ -> ...) ()
```

Parallel program safety

Parallel program safety

- Replicated coherency

Why ?



Replicat

Type system

Parallel program safety

- Replicated coherency
- Level (memory) compatibility

Level(memory) compatibility

```
<< let multi f x = ... >>
let x = #y#
let z = $v$
```

Type system

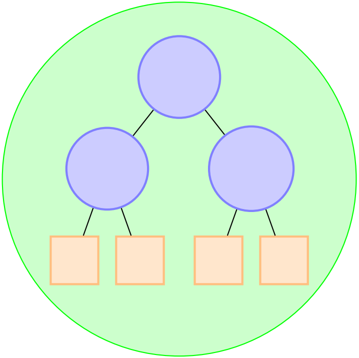
Parallel program safety

- Replicated coherency
- Level (memory) compatibility
- Control parallel structure nesting
  - vector
  - tree

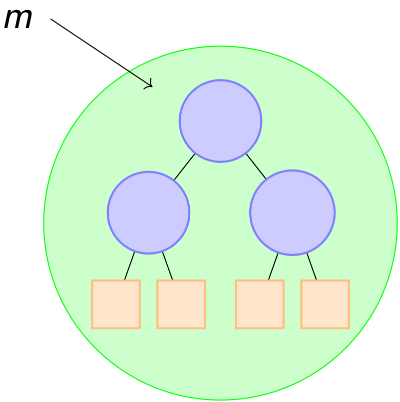
Parallel structure nesting

```
<< let v = << 1 >> in v >>
let v = << 1 >> in << v >>
```

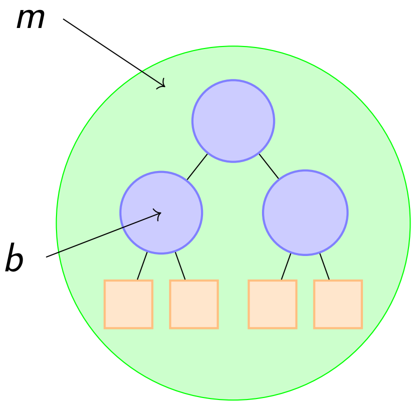
Type localities



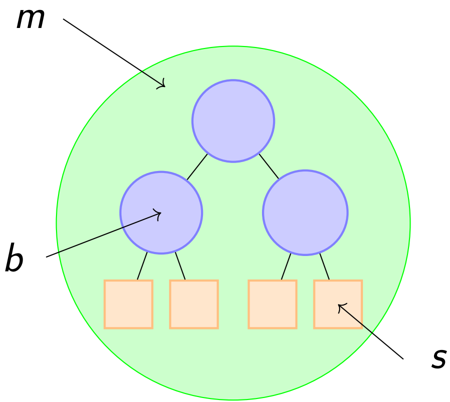
Type localities



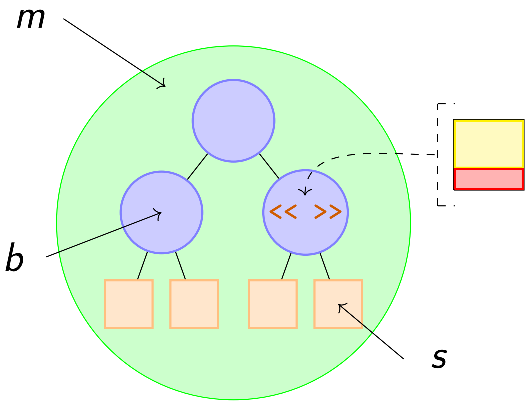
Type localities



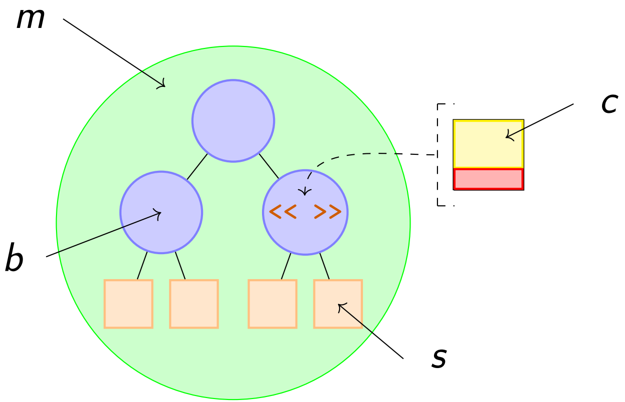
Type localities



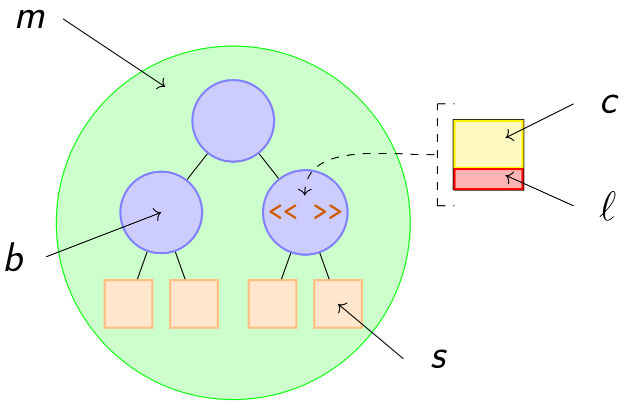
Type localities



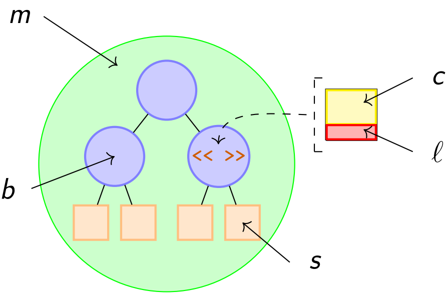
Type localities



Type localities



Type annotations



Type grammar

$$
\begin{array}{lll}
\tau & ::= & \\
 & \alpha_\pi & \textit{type variable} \\
 & \texttt{Base}_\pi & \textit{base type} \\
 & (\tau,\tau)_\pi & \textit{pair} \\
 & \tau\ \texttt{Par}_b & \textit{vector} \\
 & \tau\ \texttt{Tree}_\pi & \textit{tree} \\
 & (\tau \xrightarrow{\pi} \tau)_\pi & \textit{arrow type}
\end{array}
$$

$$\pi ::= m \mid b \mid c \mid l \mid s$$

Type annotations

Latent effect

$$(\tau \xrightarrow{\pi} \tau)_{\pi'}$$

Where π is the effect *emitted* by the evaluation and π' the locality of definition.

A BSP function

```
#let f = fun x ->
  let v = << ... >> in 1
-: val f : ('a_`z -(b)-> int_b)_m
```

$$f : ({}'a_{`z} \xrightarrow{b} int_b)_m$$

Accessibility: $\lhd$

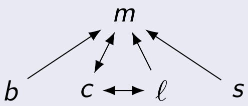
$$
\begin{array}{rcl}
m, c & \lhd & m \\
m, b & \lhd & b \\
m, l, c & \lhd & l \\
m, l, c & \lhd & c \\
m, s & \lhd & s
\end{array}
$$

$$
\begin{tikzcd}
 & & m & & \\
b \arrow[urr] & c \arrow[ur, leftrightarrow] \arrow[rr, leftrightarrow] & & \ell \arrow[ul] & s \arrow[ull]
\end{tikzcd}
$$

$\lambda_2 \lhd \lambda_1$: « λ_1 *can* read *in* λ_2 *memory.* »

Accessibility: $\lhd$

$$
\begin{array}{rcl}
m, c & \lhd & m \\
m, b & \lhd & b \\
m, l, c & \lhd & l \\
m, l, c & \lhd & c \\
m, s & \lhd & s
\end{array}
$$



$\lambda_2 \lhd \lambda_1$: « λ_1 *can* read *in* λ_2 *memory.* »

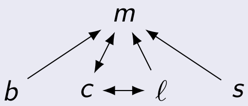
Example:

$$f : ({}'a_z \xrightarrow{b} int_b)_m$$

$$\texttt{f 1} \rightsquigarrow b \lhd m$$

Accessibility: $\lhd$

$$
\begin{array}{rcl}
m, c & \lhd & m \\
m, b & \lhd & b \\
m, l, c & \lhd & l \\
m, l, c & \lhd & c \\
m, s & \lhd & s
\end{array}
$$



$\lambda_2 \lhd \lambda_1$: « λ_1 *can* read *in* λ_2 *memory.* »

Example:

$$f : ('a_z \xrightarrow{b} int_b)_m$$

`f 1` $\rightsquigarrow b \lhd m$

Error

Definability

Definability: $\blacktriangleleft$

$$
\begin{array}{rcl}
s, b, m & \blacktriangleleft & m \\
b & \blacktriangleleft & b \\
l, c & \blacktriangleleft & c \\
l, c & \blacktriangleleft & l \\
s & \blacktriangleleft & s
\end{array}
$$

$$
\begin{tikzcd}
 & & m & \\
b \arrow[urr] & c \arrow[r, leftrightarrow] & \ell \arrow[u] & s \arrow[ul]
\end{tikzcd}
$$

$\lambda_1 \blacktriangleleft \lambda_2$: « λ_1 *can be* defined *in* λ_2 *memory.* »

Definability: $\blacktriangleleft$

$$
\begin{array}{rcl}
s, b, m & \blacktriangleleft & m \\
b & \blacktriangleleft & b \\
l, c & \blacktriangleleft & c \\
l, c & \blacktriangleleft & l \\
s & \blacktriangleleft & s
\end{array}
$$

$$
\begin{tikzcd}
 & & m & \\
b \arrow[urr] & c \arrow[r, leftrightarrow] & \ell \arrow[u] & s \arrow[ul]
\end{tikzcd}
$$

$\lambda_1 \blacktriangleleft \lambda_2$: « λ_1 *can be* defined *in* λ_2 *memory.* »

Example:

`<< let multi f x = ... >>` $\rightsquigarrow m \blacktriangleleft c$

Definability: ◀

$$
\begin{array}{rcl}
s, b, m & \blacktriangleleft & m \\
b & \blacktriangleleft & b \\
l, c & \blacktriangleleft & c \\
l, c & \blacktriangleleft & l \\
s & \blacktriangleleft & s
\end{array}
$$

$$
\begin{tikzcd}
 & & m & \\
b \arrow[urr] & c \arrow[r, leftrightarrow] & \ell \arrow[u] & s \arrow[ul]
\end{tikzcd}
$$

$\lambda_1 \blacktriangleleft \lambda_2$: « λ_1 *can be* defined *in* λ_2 *memory.* »

Example:

`<< let multi f x = ... >>` $\rightsquigarrow m \blacktriangleleft c$

Error

Other relations

Propagation

This relation returns the prevailing effect among ε and ε'.

$$\begin{tikzcd}
\mathtt{s} \arrow[r] \arrow[d] & \mathtt{m} & \\
\mathtt{b} \arrow[ur] & \mathtt{l} \arrow[u] \arrow[l] & \mathtt{c} \arrow[ul] \arrow[l] \arrow[ll, bend left]
\end{tikzcd}$$

Serialisation

Is it safe to communicate τ_π to locality Λ ?

$$\begin{array}{lcl}
\mathbf{Seria}_\Lambda(\mathtt{Base}_\pi) & = & \mathtt{Base}_\Lambda \;\; \textit{if } \mathtt{Base} = \mathtt{int}, \mathtt{string}, \mathtt{float}, \mathtt{Bool}, \ldots \\
\mathbf{Seria}_\Lambda(\mathtt{Base}_\pi) & = & \mathtt{Fail} \;\; \textit{if } \mathtt{Base} = \mathtt{i/o}, \ldots \\
\mathbf{Seria}_\Lambda(\tau_\pi) & = & \begin{cases} \tau_\Lambda, & \textit{if } \pi \lhd \Lambda \\ \mathtt{Fail}, & \textit{otherwise} \end{cases} \\
\mathbf{Seria}_\Lambda(\tau_\pi \; \mathtt{par}_\mathtt{b}) & = & \mathtt{Fail}
\end{array}$$

Operational semantics

Operational semantics

- Big Step semantics (deterministic)

Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)

Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
- Programs that *"do not go wrong"* : $(\exists v. \Downarrow_p^{\mathcal{L}} v)$ or $(\Downarrow_p^{\mathcal{L}} \infty)$

Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
- Programs that *"do not go wrong"* : $(\exists v. \Downarrow_p^{\mathcal{L}} v)$ or $(\Downarrow_p^{\mathcal{L}} \infty)$

Type safety of a MULTI-ML program

- Let e be an expression,
- Γ a typing environment,
- and c a set of constraint.

Then: $\Gamma \vdash e : \tau_\pi/\varepsilon[c]$ implies that e *"does not go wrong"* $(e \Rightarrow_{safe})$

Table of Contents

1 Introduction

2 The MULTI-ML language

3 Type system

4 Implementation

Execution scheme

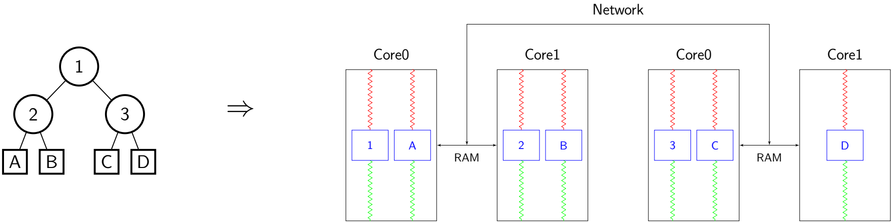
Parallel and sequential implementations

Benchmarks

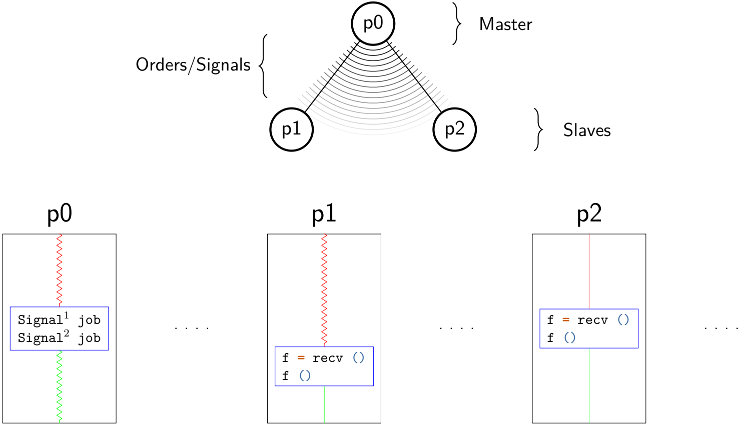
5 Conclusion

Execution scheme

- One process per leaf/node
- Distributed over physical cores



Execution scheme



Formal properties

Correctness of a MULTI-ML program

If $e \Rightarrow_{safe}$ and $\mathcal{WF}(e)$ we have : $\langle\langle [\![e]\!]_{\mathrm{M}}, ..., [\![e]\!]_{\mathrm{M}} \rangle\rangle \Rightarrow_{safe}$

Cost model

Big step semantics with costs

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} e' / C_p} \text{ and } \frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathsf{b}} e' / C_p / < C_{p.i} >}$$

Cost model

Big step semantics with costs

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} e' / C_p} \quad \text{and} \quad \frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathsf{b}} e' / C_p / < C_{p.i} >}$$

Cost algebra

C	$::=$		
	$\mid$	1	*Arbitrary unit cost*
	$\mid$	$\mathbf{g}$	*g parameter*
	$\mid$	$\mathbf{l}$	*l parameter*
	$\mid$	$C \oplus C$	*Addition*
	$\mid$	$C \otimes C$	*Multiplication*
	$\mid$	$\max\limits_{i=0}^{n}(C_i)$	*Maximum*
	$\mid$	$\sum\limits_{i=0}^{n}(C_i)$	*Sum*
	$\mid$	$\mathcal{S}_\varepsilon(v)$	*Data size*

$$\begin{array}{lcl}
C_1 \oplus C_2 & \equiv & C_2 \oplus C_1 \\
C_1 \otimes C_2 & \equiv & C_2 \otimes C_1 \\
\max\limits_{i=0}^{n}(C_i) & \equiv & maximum(C_0, ..., C_n) \\
\sum\limits_{i=0}^{n}(C_i) & \equiv & C_0 \oplus ... \oplus C_n
\end{array}$$

Distributed implementation

Module

- Communication library
- Based on operational semantics

Current implementation

- MPI processes
- Distributed over physical cores
- Shared/Distributed memory

Future implementations

- TCP/IP
- PTHREAD
- ...

Sequential implementation

Sequential simulator

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

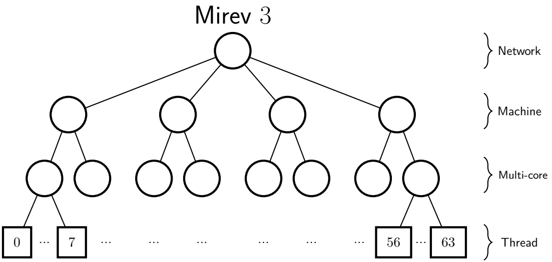
```
#let multi tree f n =
  where node =
     let r =<<f ($pid$ + #n# + 1) >> in
       finally r (gid^"=>"^n)
  where leaf=
     (gid^"=>"^n);;

- : 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"
```

Benchmarks

Naive Eratosthenes sieve

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

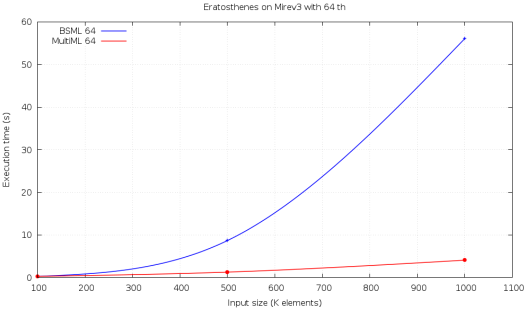
Benchmarks



Naive Eratosthenes sieve

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

Benchmarks



Naive Eratosthenes sieve

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

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

Table of Contents

1 Introduction

2 The MULTI-ML language

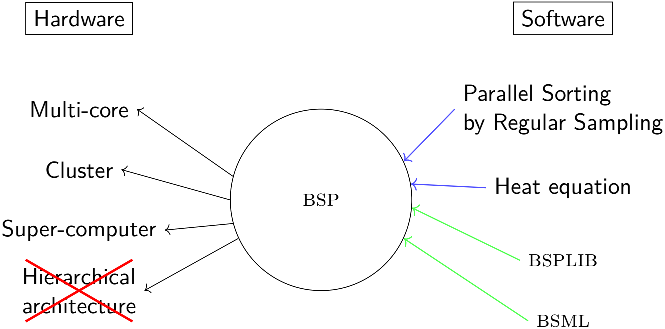
3 Type system

4 Implementation

5 Conclusion

Before ...

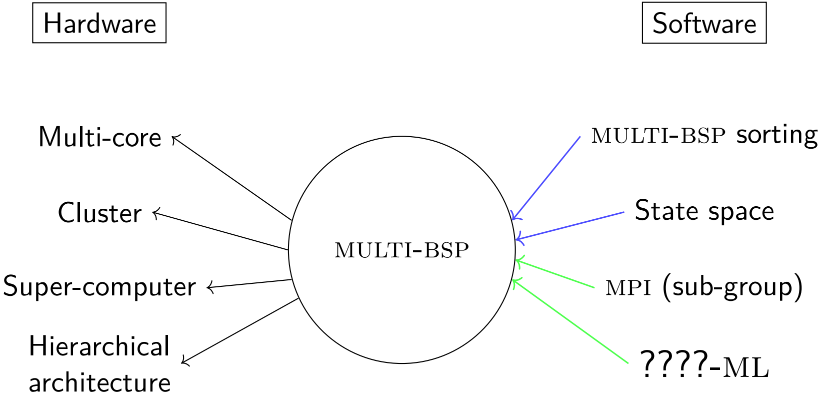
- BSP $\neq$ Hierarchical architecture
- BSML $\rightarrow$ BSP à la ML
- No language dedicated to MULTI-BSP



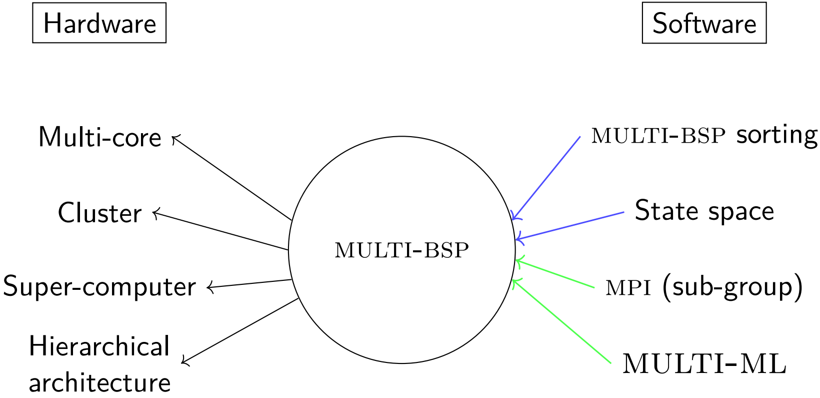
... Now

- MULTI-BSP extension of ML
  - Recursive multi-functions `let multi f x = ...`
  - BSML like code `where node = << f ... >>`
  - Small syntax extension `#,$,at,mkpar,finally,mktree,...`
- Type system
  - Constraints
  - Effects
- Operational semantics (even for diverging terms)
- Compilation scheme
- $\Rightarrow$ Type safety from programs to abstract machines

Before ...


Hardware
Software
Multi-core
Cluster
Super-computer
Hierarchical architecture
MULTI-BSP
MULTI-BSP sorting
State space
MPI (sub-group)
????-ML

... Now



Hardware
Software
Multi-core
Cluster
Super-computer
Hierarchical architecture
MULTI-BSP
MULTI-BSP sorting
State space
MPI (sub-group)
MULTI-ML

Future Work

Ongoing work

- Code examples
  - FFT, TDS, PPP, Sort, Nbody, State-Space, MM, ...
- Extensions
  - Language
  - Type system
- MULTI-ML + GPU $\Rightarrow$ Hybrid architectures

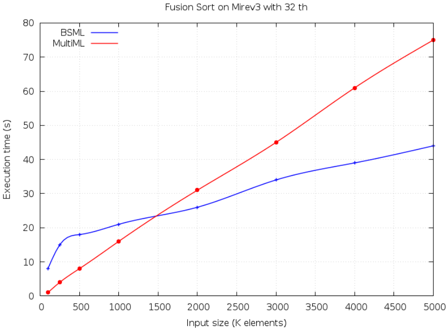
Future work

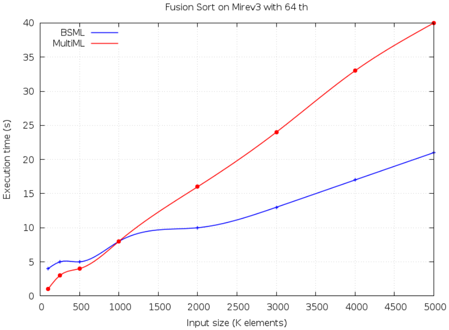
- Automatic cost analysis
- Certified parallel programming

Thank you for your attention ☺

Questions ?

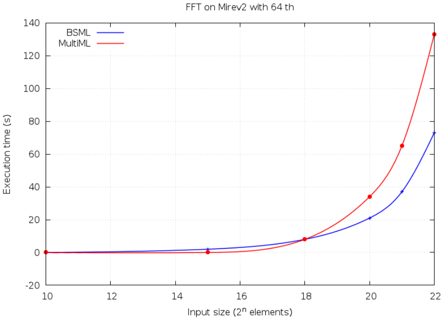
Fusion Sort on Mirev3 with 32 and 64 threads

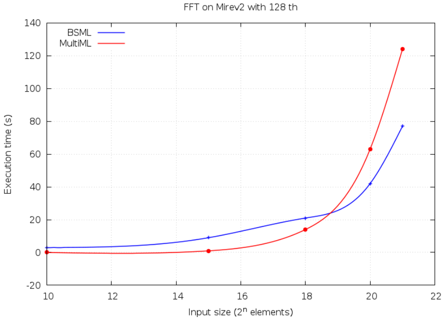




Fast Fourier Transform

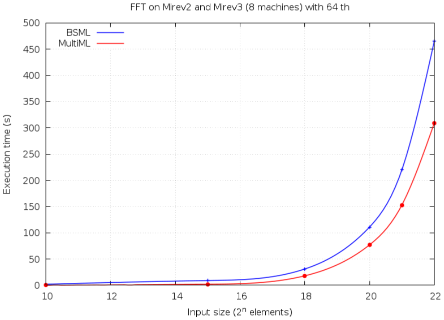
FFT on Mirev2 (8 machines) with 64 and 128 threads

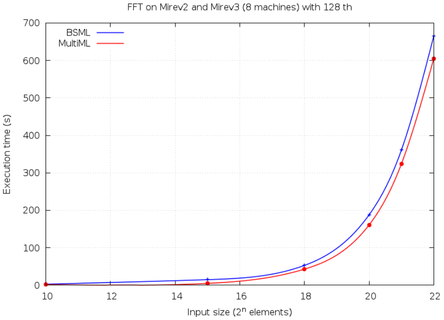




Fast Fourier Transform

FFT on Mirev2 (6 machines) and Mirev3 (2 machines) with 64 and 128 threads





Typing rules

$$\text{LET IN}\quad \frac{\begin{array}{l}\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1} / \varepsilon_1 \ \ [c_1] \\ \Lambda, \Gamma ; x : \mathbf{Weak}(\tau^1_{\pi_1}, \varepsilon_1) \vdash e_2 : \tau^2_{\pi_2} / \varepsilon_2 \ \ [c_2] \\ c_3 \equiv [\Psi = \mathbf{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2]\end{array}}{\Lambda, \Gamma \vdash \mathbf{let}\ \ x = e_1\ \ \mathbf{in}\ \ e_2 : \tau^2_{\pi_2} / \Psi \ \ [c_3]}$$

```
<< fun _ -> let x = at t in x >>
```

```
<< let x = at t in fun _ -> x >>
```