FUNCTIONAL ABSTRACTION FOR PROGRAMMING
MULTI-LEVEL ARCHITECTURES:
FORMALISATION AND IMPLEMENTATION

VICTOR ALLOMBERT

under the supervision of:
FRÉDÉRIC GAVA and JULIEN TESSON

PH.D. DEFENSE
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
   Parallel programming models

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
Parallel computing over the years

1970-80

1990-00

2010-now

The beginning
Parallel computing over the years

1970-80

Cray-1

1990-00

Shared memory

2010-now

The beginning
Parallel computing over the years

1970-80
- The beginning
- Cray-1

1990-00

2010-now

Shared memory
Shared memory models

Characterised by:

- A shared memory
- Integrated network (NUMA)
- OPENMP/PTHREAD (C, FORTRAN)
Parallel computing over the years

- **1990-00**: Distributed memory clusters.
- **2010-now**: Shared memory.
Parallel computing over the years

The beginning

Shared memory

Cray-1

1970-80

1990-00

2010-now

Distributed memory

Clusters
Parallel computing over the years

Shared memory

1970-80

1990-00

2010-now

Cray-1

Clusters

The beginning

Distributed memory
Distributed computing

Characterised by:

- Interconnected units
- Distributed memory
- Communication network
- MPI/map-reduce
Parallel computing over the years

- Shared memory
- Distributed memory

The beginning
Cray-1
Clusters

1970-80
1990-00
2010-now

V. Allombert
7 July 2017
Parallel computing over the years

- **The beginning**
  - Shared memory
  - Cray-1

- **1970-80**
  - Distributed memory

- **1990-00**
  - Hierarchical memory

- **2010-now**
  - Clusters

**Roadrunner**

**T aihuLight**
Parallel computing over the years

1970-80
- The beginning
- Cray-1

1990-2000
- Clusters

2010-now
- Roadrunner

Shared memory

Distributed memory

Hierarchical memory
Parallel computing over the years

1970-80
- The beginning
- Cray-1

1990-00
- Clusters

2010-now
- Roadrunner
- TaihuLight

Shared memory
Distributed memory
Hierarchical memory
Hierarchical architectures

Characterised by:

- Interconnected units
- Both shared and distributed memories
- Hierarchical memories
Parallel computing over the years

1970-80
- The beginning
- Cray-1

1990-00
- Clusters

2010-now
- Roadrunner
- TaihuLight

Shared memory

Distributed memory

Hierarchical memory
Parallel programming models

- Implicit
- Explicit

Automatic Parallelisation:

+ Easy
+ Transparent
  - Limited
  - “Naive”

- Par4All
- Intel C++ compiler
- Vienna Fortran compiler
Parallel programming models

- Parallel programming
  - Implicit
    - Automatic Parallelisation
  - Explicit
    - Skeletons

Skeletons:
- Easy
- Structured
  - Difficult to extend
- Cost model
  - SKML
  - SKETO
  - Muesli
Parallel programming models

- Parallel programming
  - Implicit
    - Automatic Parallelisation
  - Explicit
    - Skeletons
    - Data Parallelism

Data Parallelism:

- Structured
- Patterns
  - Limited
  - Complex
- OPENMP
- SAC
- CUDA
Parallel programming models

Parallel programming

Implicit
- Automatic Parallelisation
- Skeletons

Explicit
- Data Parallelism
- Concurrent Programming

Concurrent Programming:
- Flexible
- Powerful
- Complex
- Error prone

- MPI
- PTHREAD
- ERLANG/JOCAML
Why structured parallelism?

Pieces (Data)  Workers (Processes)  House (Results)
Why structured parallelism?

Pieces (Data)  Workers (Processes)  House (Results)
Why structured parallelism?

Structured

Pieces (Data) + Workers (Processes) → House (Results)

V. Allombert
Why structured parallelism?

structured +

Pieces (Data)  Workers (Processes)  House (Results)
Why structured parallelism?

Pieces (Data) + Workers (Processes) → House (Results)
Why structured parallelism?

Pieces (Data) + Workers (Processes) → House (Results)
Why structured parallelism?

Structured + Unstructured \rightarrow House (Results)

Pieces (Data) + Workers (Processes)
Why structured parallelism?

Structured

+ 

Workers (Processes)

→ 

House (Results)

Pieces (Data)

UNSTRUCTURED

STRUCTURED
Why structured parallelism?

Structured + Unstructured → House (Results)

Pieces (Data) + Workers (Processes) → House (Results)
Why structured parallelism?

 pieces (data) + workers (processes) → house (results)

structured

unstructured

V. Allombert 7 July 2017
A sequential bridging model

Hardware

- x86
- x64
- ARM
- PowerPC

Von Neumann

Software

- Quick Sort
- Compiler X
- ML
- C
A parallel bridging model

Hardware
- Multi-Core
- Cluster
- Super-computer
- GPU
- FPGA

Software
- Sorting algorithms
- Compilers
- Language
- Skeletons
A parallel bridging model

Hardware

- Multi-core
- Cluster
- Super-computer

Software

- Parallel Sorting by Regular Sampling
- Heat equation
- BSPLIB
- BSML
**Bulk Synchronous Parallelism**

### The BSP computer

Defined by:

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

Properties:

- **Deadlock-free**
- **Predictable performances**
Bulk Synchronous Parallelism

The BSP computer

Defined by:
- $p$ pairs CPU/memory

...
Bulk Synchronous Parallelism

The BSP computer

Defined by:

- \( p \) pairs CPU/memory
- Communication network
**Bulk Synchronous Parallelism**

The **BSP** computer

Defined by:
- $p$ pairs CPU/memory
- Communication network
- Synchronisation unit
**Bulk Synchronous Parallelism**

### The BSP computer

Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

![Diagram showing local computations, communication, barrier, and next super-step between processes](image)
Bulk Synchronous Parallelism

The BSP computer

Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:
- Deadlock-free
- Predictable performances

Diagram:
- Local computations
- Communication barrier
- Next super-step
**Bulk Synchronous Parallelism**

The **BSP** computer

Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:
- Deadlock-free

![Diagram of BSP computer architecture]

- \( p_0 \)
- \( p_1 \)
- \( p_2 \)
- \( p_3 \)

local computations
communication
barrier
next super-step
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

Hardware
- Multi-core
- Cluster
- Super-computer

Software
- Parallel Sorting by Regular Sampling
- Heat equation
- BSPLIB
- BSML
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
Bulk Synchronous ML

What is BSML?

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

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:

Replicated part (BSP) → parallel vector

Sequential part
A parallel bridging model

Hardware
- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software
- Parallel Sorting by Regular Sampling
- Heat equation
- BSPLIB
- BSML
A parallel bridging model

Why?
- Flat memories
- No sub-synchronisation

Hardware
- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software
- Parallel Sorting by Regular Sampling
- Heat equation
- BSPLIB
- BSML
A parallel bridging model

Hardware

- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software

- MULTI-BSP sorting
- State space
- MPI (sub-group)
- ???-ML
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

V. Allombert
7 July 2017
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

Stage 3
- Network
  - RAM
    - $P_3 = 4$

Stage 2
- RAM
  - L3
    - L2/L1
      - Core
    - $P_1 = 8$
  - $P_2 = 4$
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
BSP vs. MULTI-BSP

- **BSP**
  - Network
  - Nodes: p0, p1, p2, p3

- **Multi-BSP**
  - Core
  - Time

V. Allombert

7 July 2017

21 / 45
BSP vs. MULTI-BSP

V. Allombert

7 July 2017
The **MULTI-BSP** model

**Execution model**

A level $i$ superstep is:

- Level $i$ executes code independently
- Exchanges information with the memory
- Synchronises
The **MULTI-BSP** model

**Execution model**

A level $i$ superstep is:

- Level $i - 1$ executes code independently
The MULTI-BSP model

Execution model

A level $i$ superstep is:

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

```
let v = <<e>>
```

```
<< e ... e >>
```

```
// Replicated part (BSP)
```

```
// Sequential part
```

```
parallel vector
```

V. Allombert

7 July 2017

23 / 45
The MULTI-ML language

Basic ideas

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

\[
\text{let } v = \langle\langle e \rangle\rangle \\
\langle\langle e \rangle\rangle \rightarrow \{(f_0, f_1, \ldots, f_{p-1})\} \quad \text{parallel vector}
\]

\[
\text{Sequential part}
\]

V. Allombert 7 July 2017 23 / 45
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

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

Recursion structure

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

V. Allombert 7 July 2017 24 / 45
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]
MULTI-ML: Tree recursion

Recursion structure

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

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

Result
Recursion structure

\[
\begin{align*}
\text{let} & \ \text{multi f [args]} = \\
& \quad \text{where node} = \\
& \quad \quad (* \text{ BSML code } *) \\
& \quad \quad ... \\
& \quad \quad \text{<< f [args] >>} \\
& \quad \quad ... \ \text{in v} \\
& \quad \text{where leaf} = \\
& \quad \quad (* \text{ OCaml code } *) \\
& \quad \quad ... \ \text{in v}
\end{align*}
\]

\[
\begin{array}{c}
\text{Result: } v_{0.0}, v_{0.1}, v_{0.0.0}, v_{0.0.1}, v_{0.1.0}, v_{0.1.1}
\end{array}
\]
MULTI-ML: Tree recursion

Recursion structure

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

Diagram:

```
  v0.0
 /   \
/     /
/       /
v0.1
 /   \
/     /
```

V. Allombert 7 JULY 2017 24 / 45
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

```
\[ \uparrow v_0 \]
```
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**

```
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

```ocaml
define 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

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

![Tree diagram]

V. Allombert

7 July 2017
**MULTI-ML: Tree construction**

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

Summary

- `mktree e`
Primitives

Summary

- mktree e
- gid

![Diagram](attachment://tree.png)
Primitives

Summary

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

Summary

- mktree e
- gid
- at

\[
\begin{array}{c}
V_0.0 \\
V_0.1 \\
V_1.0 \\
V_1.1 \\
\end{array}
\]
Primitives

Summary

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

Summary

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

Summary

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

```
x let x = ...
```

![Diagram](attachment:diagram.png)
Primitives

Summary

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

Diagram:

```
x  let x = ...

<< #x# >> #x#
```

V. Allombert
Primitives

Summary

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

```
let x = ...
```
Primitives

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

```
mkpar (fun i -> vi)
```
Primitives

Summary

- \texttt{mktree e}
- \texttt{gid}
- \texttt{at}
- \texttt{<<...f...>>}
- \texttt{#x#}
- \texttt{mkpar f}

\begin{tikzpicture}
  \node (mkpar) {mkpar (fun i -> vi)};
  \node (f0) [below left of=mkpar] {f 0; f 1};
  \node (f1) [below right of=mkpar] {mkpar (fun i -> vi)};
  \draw (mkpar) -- (f0);
  \draw (mkpar) -- (f1);
\end{tikzpicture}
Primitives

Summary

- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`
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
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
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
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
```
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 finally rc s  
  where leaf =  
    sumSeq l
```

V. Allombert 7 July 2017 27 / 45
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 finally rc s
  where leaf =
    sumSeq l
```

![Diagram of a tree with nodes labeled with intervals [0;1], [2;3], [4;5], [6;7].]
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
```
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
```
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
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 finally rc s
  where leaf =
    sumSeq l
```

Code example
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 finally rc s
  where leaf = 
    sumSeq l
```

V. Allombert
7 JULY 2017
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
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 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

```latex
if random_bool () then
  multi_fct ()
else
  (fun _ -> ...) ()
```
Parallel program safety

- Replicated coherency

Why?

Replicated coherency
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 imbrication
  - vector
  - tree

Parallel structure imbrication

```plaintext
<< let v = << 1 >> in v >>
let v = << 1 >> in << v >>
```
Type localities
Type localities

$m$
Type localities

$m$

$b$

V. Allombert
7 July 2017
Type localities
Type localities
Type annotations

Type grammar

\[ \begin{align*}
\tau \quad &::= \\
\alpha &_{\pi} \quad \text{type variable} \\
\Base &_{\pi} \quad \text{base type} \\
(\tau, \tau) &_{\pi} \quad \text{pair} \\
\tau \Par &\_b \quad \text{vector} \\
\tau \Tree &\_\pi \quad \text{tree} \\
(\tau \rightarrow \tau) &_{\pi} \quad \text{arrow type}
\end{align*} \]

\[ \pi \quad ::= \quad m \mid b \mid c \mid l \mid s \]
Type annotations

Latent effect

\[(\mathcal{T} \xrightarrow{\pi} \mathcal{T})_{\pi'}\]

Where \(\pi\) is the effect emitted by the evaluation and \(\pi'\) the locality of definition.

A BSP function

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

f : ('a'z \rightarrow b \rightarrow int_b) _m```

V. Allombert
Accessibility: \( \triangleleft \)

\[
\begin{align*}
m, c & \triangleleft m \\
m, b & \triangleleft b \\
m, l, c & \triangleleft l \\
m, l, c & \triangleleft c \\
m, s & \triangleleft s
\end{align*}
\]

\[\lambda_2 \triangleleft \lambda_1 : \text{« \( \lambda_1 \) can read in \( \lambda_2 \) memory. »}\]
Accessibility:

\[
\begin{align*}
m, c &\triangle m \\
m, b &\triangle b \\
m, l, c &\triangle l \\
m, l, c &\triangle c \\
m, s &\triangle s \\
\end{align*}
\]

\[\lambda_2 \triangle \lambda_1 : \text{« \( \lambda_1 \) can read in \( \lambda_2 \) memory. »}\]

Example:

\[
f : (\langle a \cdot z \Rightarrow b \rangle)_m \\
\text{f 1} \rightsquigarrow b \triangle m
\]
Accessibility

\[ \lambda_2 \triangleleft \lambda_1 : \text{“}\lambda_1 \text{ can read in } \lambda_2 \text{ memory. ”} \]

Example:

\[ f : (\text{'a} z \rightarrow \text{int}_b)_m \]
\[ f 1 \leadsto b \triangleleft m \]

Error
Definability: ▲

\[ s, b, m \rightleftharpoons m \]
\[ b \rightleftharpoons b \]
\[ l, c \rightleftharpoons c \]
\[ l, c \rightleftharpoons l \]
\[ s \rightleftharpoons s \]

\[ \lambda_1 \rightleftharpoons \lambda_2: \text{« } \lambda_1 \text{ can be defined in } \lambda_2 \text{ memory. »} \]
Definability

$s, b, m \triangleright m$
$b \triangleright b$
$l, c \triangleright c$
$l, c \triangleright l$
$s \triangleright s$

$\lambda_1 \triangleright \lambda_2 : \text{« } \lambda_1 \text{ can be defined in } \lambda_2 \text{ memory. »}$

Example:

<< let multi f x = ... >> \rightsquigarrow m \triangleright c
Definability: ▲

\[ s, b, m \downarrow m \]
\[ b \downarrow b \]
\[ l, c \downarrow c \]
\[ l, c \downarrow l \]
\[ s \downarrow s \]

\[ \lambda_1 \downarrow \lambda_2 : \text{« } \lambda_1 \text{ can be defined in } \lambda_2 \text{ memory. } \text{»} \]

Example:

\[ \langle\langle \text{let multi f x = ... } \rangle\rangle \rightsquigarrow m \downarrow c \]
Error
Other relations

Propagation

This relation returns the prevailing effect among $\varepsilon$ and $\varepsilon'$.

![Diagram](image)

Serialisation

Is it safe to communicate $\tau_\pi$ to locality $\Lambda$?

- $\text{Seria}_\Lambda(\text{Base}_\pi) = \text{Base}_\Lambda$ if $\text{Base} = \text{int, string, float, Bool, ...}$
- $\text{Seria}_\Lambda(\text{Base}_\pi) = \text{Fail}$ if $\text{Base} = \text{i/o, ...}$
- $\text{Seria}_\Lambda(\tau_\pi) = \begin{cases} \tau_\Lambda, & \text{if } \pi \triangleleft \Lambda \\ \text{Fail}, & \text{otherwise} \end{cases}$
- $\text{Seria}_\Lambda(\tau_\pi \text{ par}_b) = \text{Fail}$
Formal properties

Operational semantics

• Big Step semantics (deterministic)
• Big Step semantics for diverging terms (mutually exclusive)
• Programs that “do not go wrong”:
  - \( v : Lp v \) or \( +Lp1 \)

Type safety of a multi-ml program

Let \( e \) be an expression, a typing environment, and \( c \) a set of constraint. Then:
\[ \vdash e : [c] \] implies that \( e \) “does not go wrong” (\( e \) safe)
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”: \( \exists v. \downarrow^{\mathcal{L}}_p v \) or \( \downarrow^{\mathcal{L}}_p \infty \)
Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
- Programs that “do not go wrong”: $(\exists v. \downarrow_p^L v)$ or $(\downarrow_p^L \infty)$

Type safety of a MULTI-ML program

- Let $e$ be an expression,
- $\Gamma$ a typing environment,
- and $c$ a set of constraints.

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

Orders/Signals

\[ p_0 \quad \{ \text{Master} \} \quad p_1 \quad p_2 \quad \{ \text{Slaves} \} \]

\[ f = \text{recv}() \]
\[ f() \]

\[ \text{Signal}^1 \text{ job} \]
\[ \text{Signal}^2 \text{ job} \]
Correctness of a \texttt{MULTI-ML} program

If $e \Rightarrow_{\text{safe}}$ and $WF(e)$ we have: $\langle\langle [e]_M, \ldots, [e]_M \rangle\rangle \Rightarrow_{\text{safe}}$
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
- ...

V. Allombert 7 July 2017
Sequential 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
    finally r (gid^"=>"^n)
  where leaf =
    (gid^"=>"^n);

- : val 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"
```
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

\[ \begin{array}{c}
\text{Mirev 3} \\
\text{Network} \\
\text{Machine} \\
\text{Multi-core} \\
\text{Thread}
\end{array} \]

\[
\begin{array}{ccccccccc}
0 & \ldots & 7 & \ldots & \ldots & \ldots & \ldots & \ldots & 56 & \ldots & 63
\end{array}
\]
Benchmarks

Naive Eratosthenes sieve

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

Results

<table>
<thead>
<tr>
<th></th>
<th>100_000</th>
<th>500_000</th>
<th>1_000_000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MULTI-ML</td>
<td>BSML</td>
<td>MULTI-ML</td>
</tr>
<tr>
<td>8</td>
<td>0.7</td>
<td>1.8</td>
<td>22.4</td>
</tr>
<tr>
<td>64</td>
<td>0.3</td>
<td>0.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

V. Allombert

7 JULY 2017
- BSP \neq \text{Hierarchical architecture}
- BSML \rightarrow \text{BSM à la ML}
- No language dedicated to \text{MULTI-BSP}
• **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**

⇒ **Type safety from programs to abstract machines**
Before ...

Hardware

Multi-core
Cluster
Super-computer
Hierarchical architecture

Software

MULTI-BSP sorting
State space
MPI (sub-group)
????-ML
... Now

Hardware

- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software

- MULTI-BSP sorting
- State space
- MPI (sub-group)
- MULTI-ML
Future Work

Ongoing work

- Code examples
- Extensions
  - Language
  - Type system

Future work

- MULTI-ML + GPU ⇒ Hybrid architectures
- 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

V. Allombert
7 July 2017
Fast Fourier Transform

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

\[ \text{LET IN} \quad \frac{\Lambda, \Gamma \vdash e_1 : \tau^1_{\pi_1} / \varepsilon_1 \quad [c_1]}{\Lambda, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau^2_{\pi_2} / \Psi \quad [c_3]} \]

\[ c_3 \equiv [\Psi = \text{Propgt}(\varepsilon_1, \varepsilon_2), c_1, c_2] \]

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

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

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