Programming in Multi-BSP: Multi-ML and its typing system

Victor Allombert, Frédéric Gava and Julien Tesson

LACL (Paris-Est) & LIFO (Orléans)

GDR LaMHA - LIFO

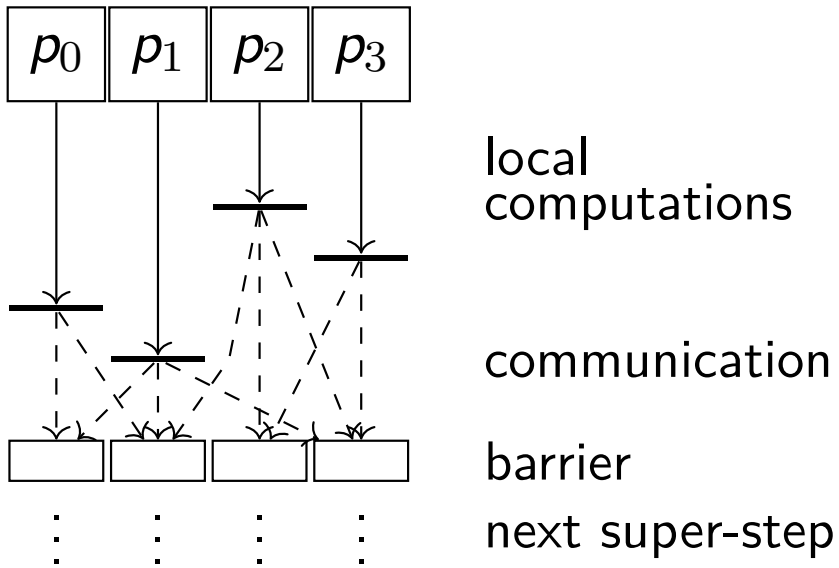
Table of Contents

Table of Contents

Bulk Synchronous Parallelism

A BSP computer:

- $\mathbf{p}$ pairs CPU/memory
- Communication network
- Synchronization unit



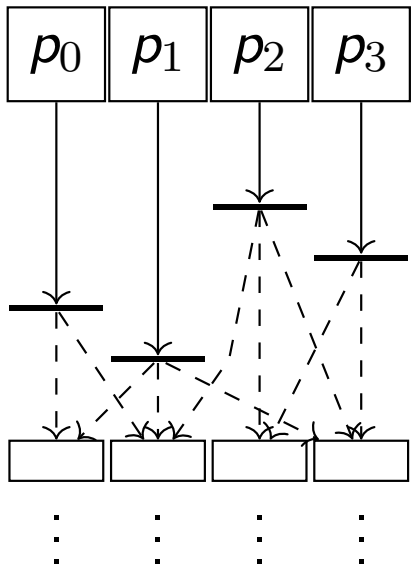
Bulk Synchronous Parallelism

A BSP computer:

- $\mathbf{p}$ pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

- Super-steps execution
- Confluent
- Deadlock-free
- Predictable performances



local computations

communication

barrier

next super-step

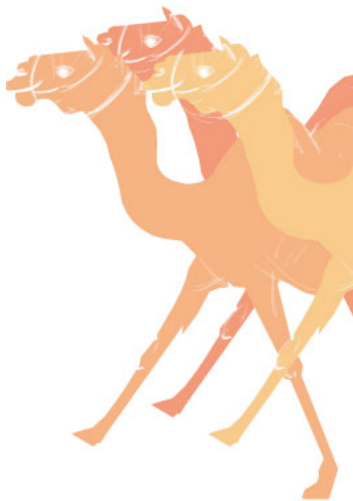
Bulk Synchronous ML

What is BSML?

Bulk Synchronous ML

What is BSML?

- Explicit BSP programming with a functional approach



Bulk Synchronous ML

What is BSML?

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

Bulk Synchronous ML

What is BSML?

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

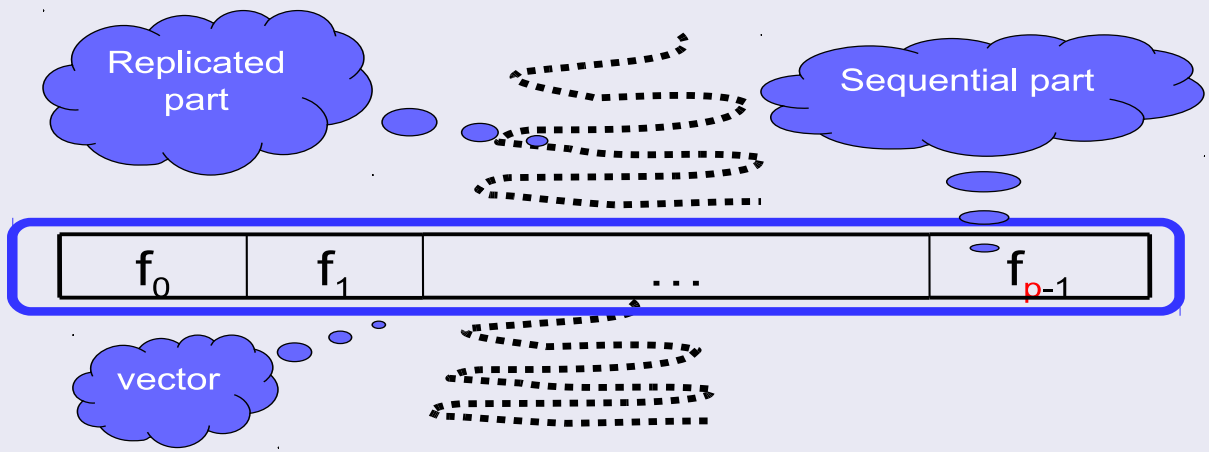
Bulk Synchronous ML

What is BSML?

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

Main idea

Parallel data structure ⇒ Vector:



BSML primitives

Asynchronous primitives

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle e, \ldots, e \rangle$

BSML primitives

Asynchronous primitives

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

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle e, \ldots, e \rangle$
- `$v$` : v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
- `$pid$` : i on processor i

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle e, \ldots, e \rangle$
- `$v$` : v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
- `$pid$` : i on processor i

Synchronous primitives

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

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle e, \ldots, e \rangle$
- `$v$` : v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
- `$pid$` : i on processor i

Synchronous primitives

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

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 = <"Hello0", "Hello1",
 ↪  "Hello2">

# let totex v = List.map (proj v) procs;;
val totex : 'a par -> 'a list = <fun>

# totex vec2;;
- : string list = ["Hello0"; "Hello1"; "Hello2"]
```

The MULTI-BSP model

What is MULTI-BSP?

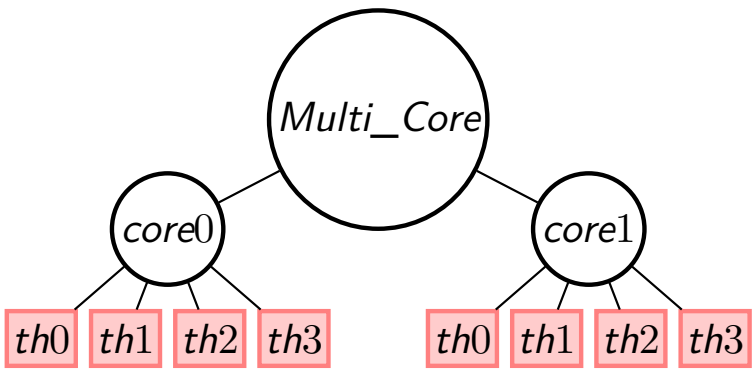
1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors

The MULTI-BSP model

What is MULTI-BSP?

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

MULTI-BSP

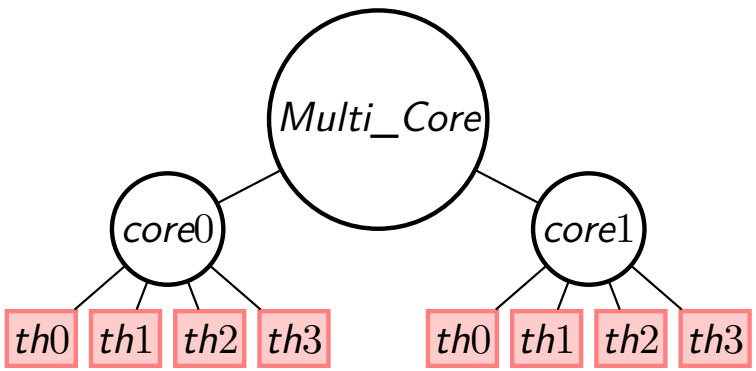


The MULTI-BSP model

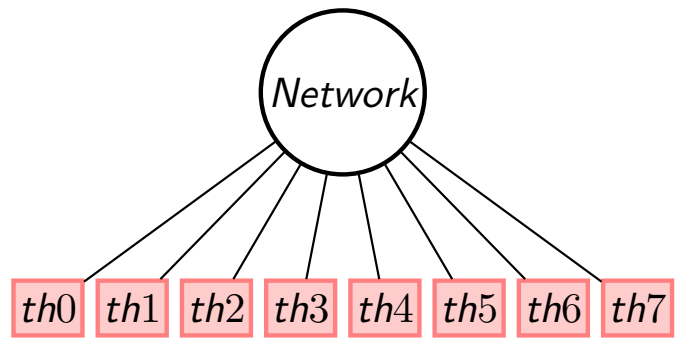
What is MULTI-BSP?

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

MULTI-BSP



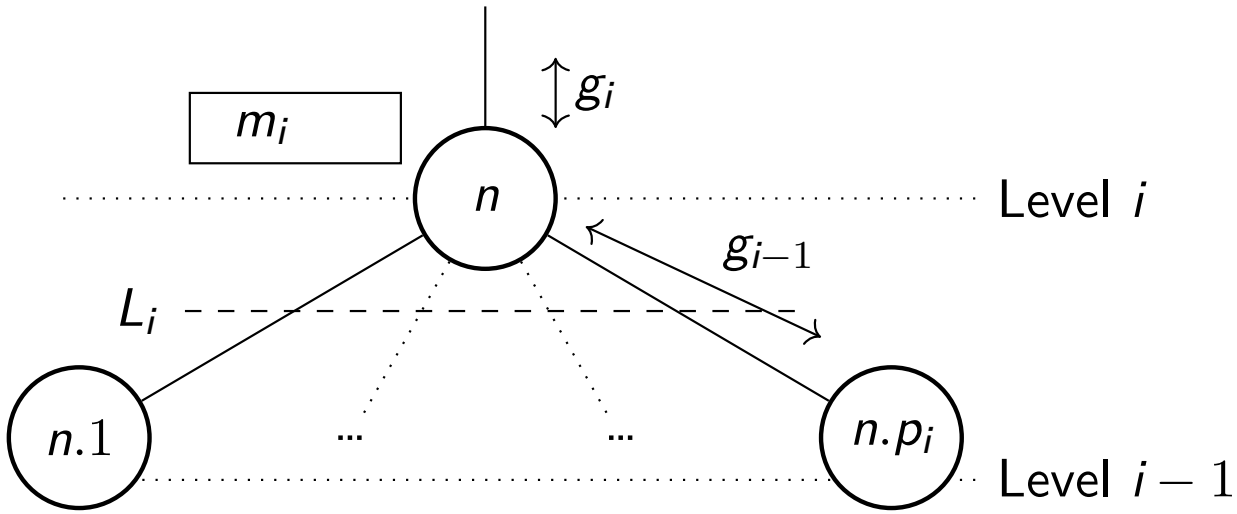
BSP



The MULTI-BSP model

Execution model

A level i superstep is:

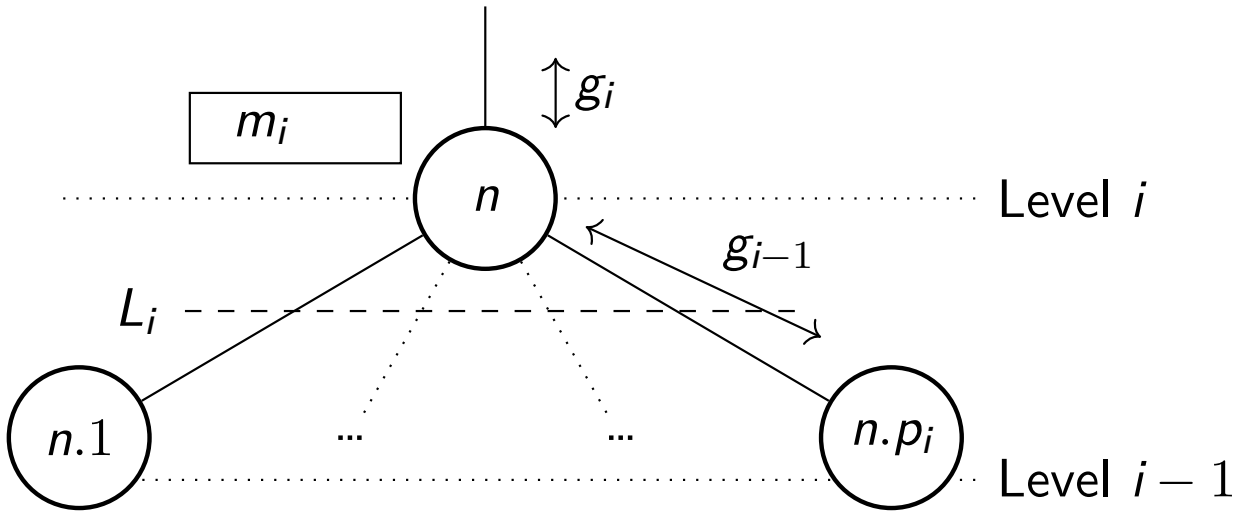


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independantly

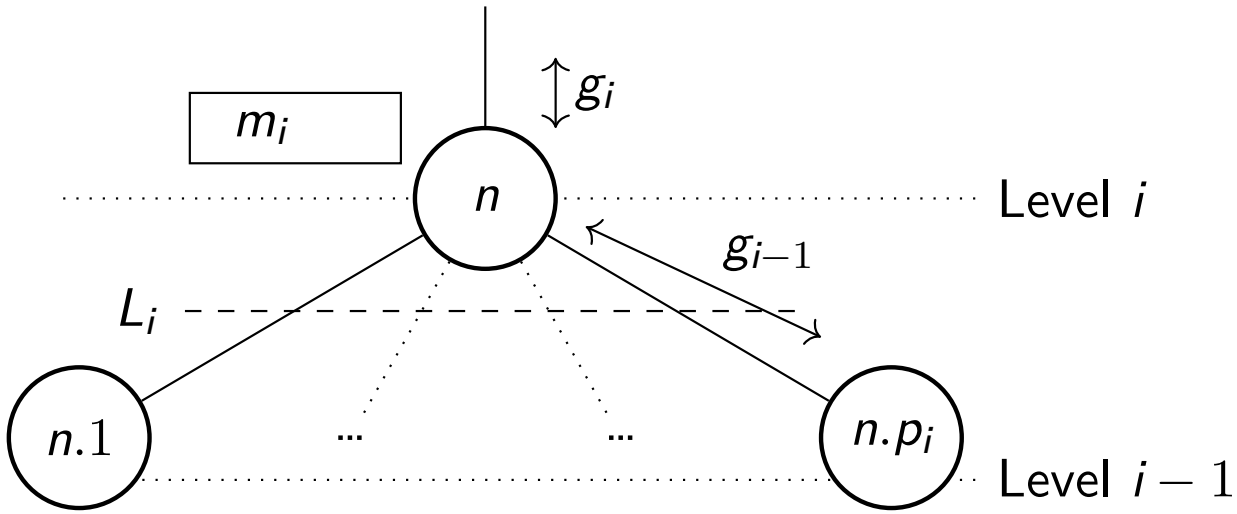


The MULTI-BSP model

Execution model

A level i superstep is:

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



The MULTI-BSP model

Execution model

A level i superstep is:

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

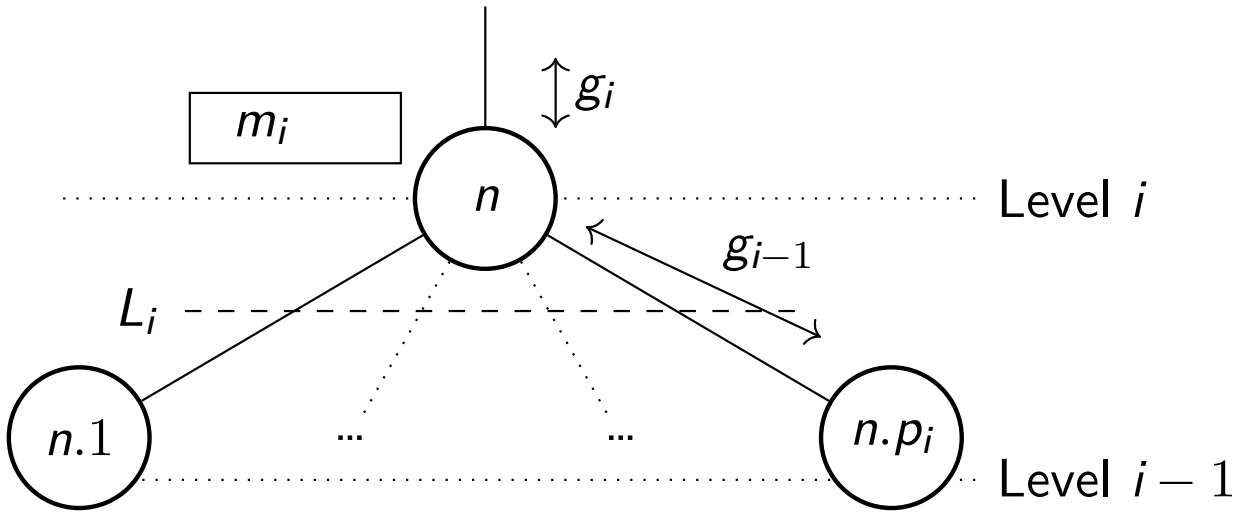


Table of Contents

Basic ideas

Basic ideas

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

Basic ideas

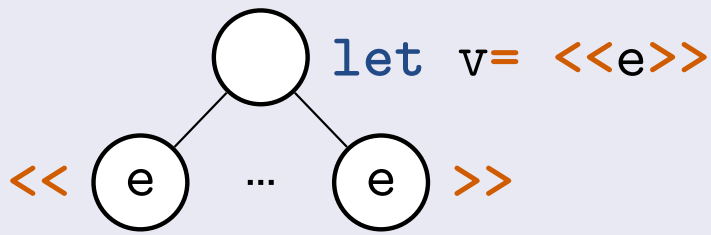
- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming

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.



MULTI-ML: Tree recursion

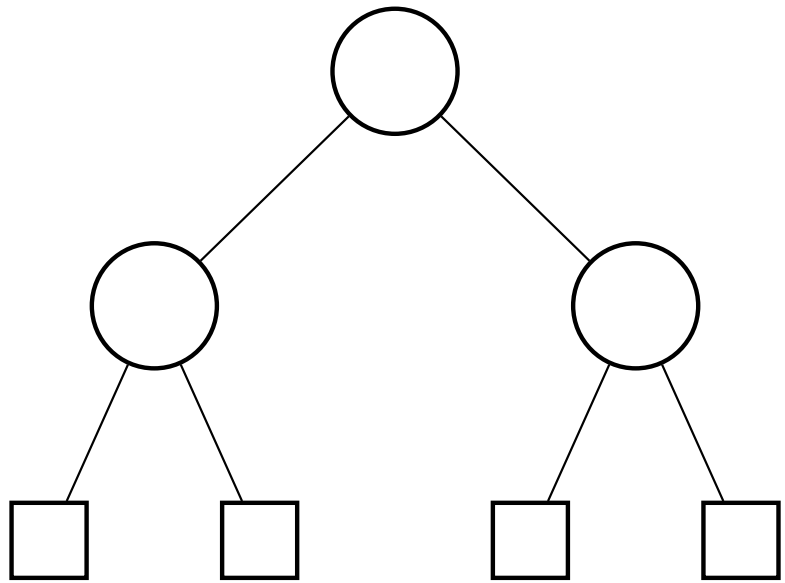
Recursion structure

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

MULTI-ML: Tree recursion

Recursion structure

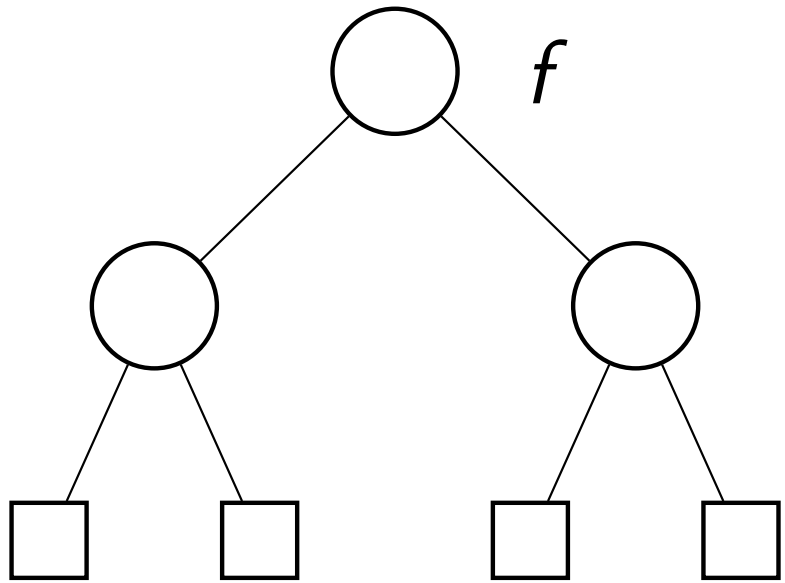
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

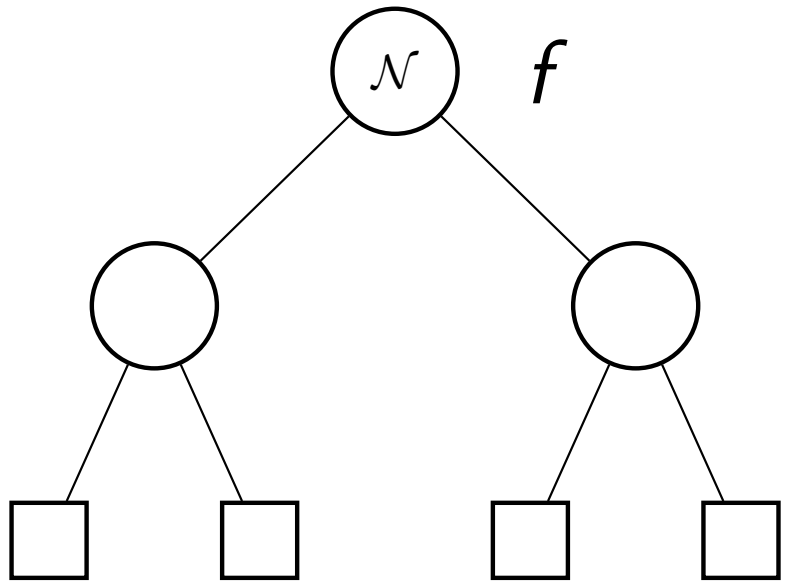
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

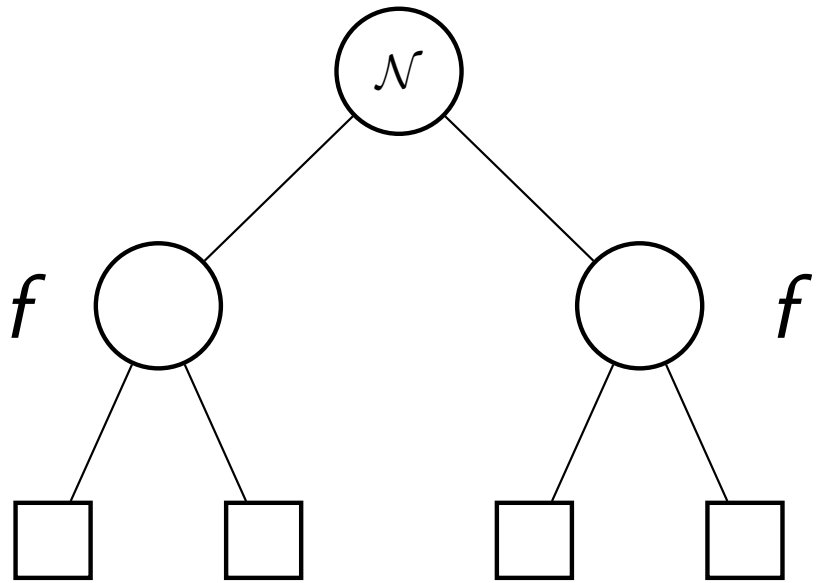
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

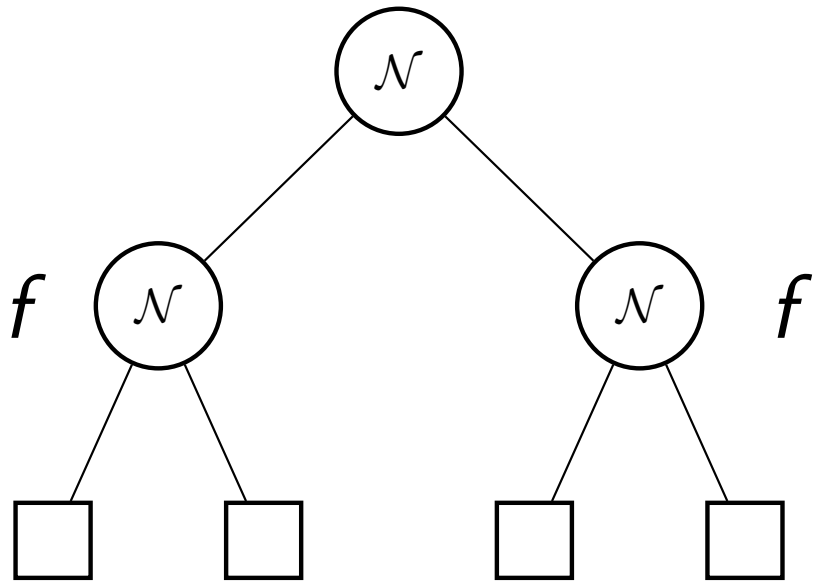
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

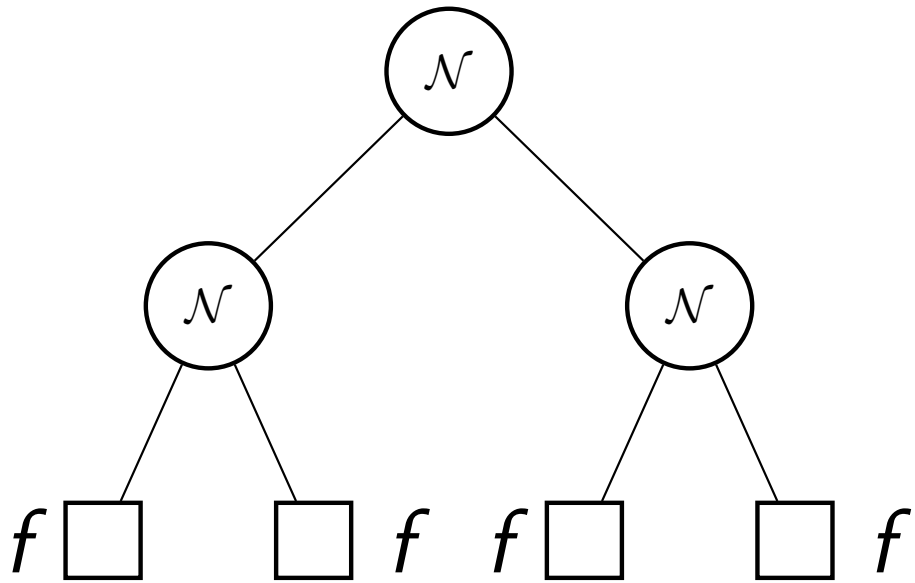
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

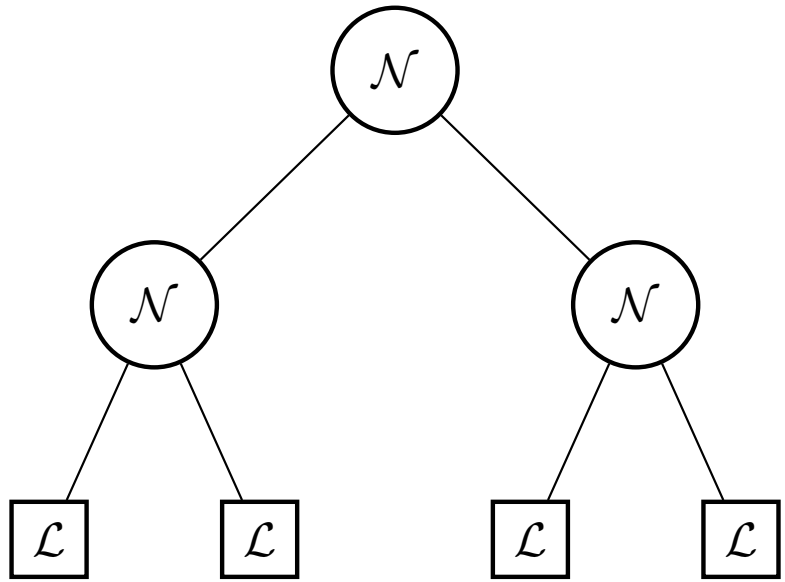
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

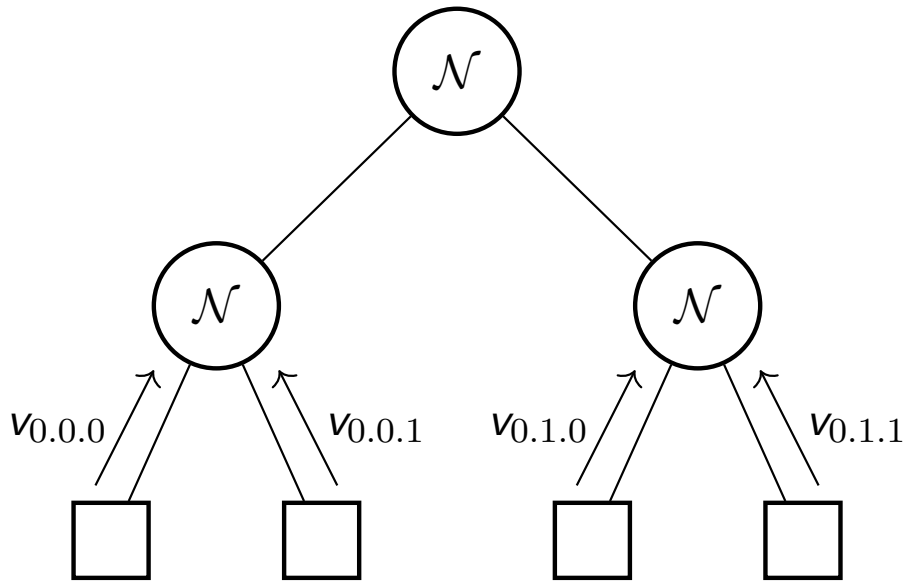
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

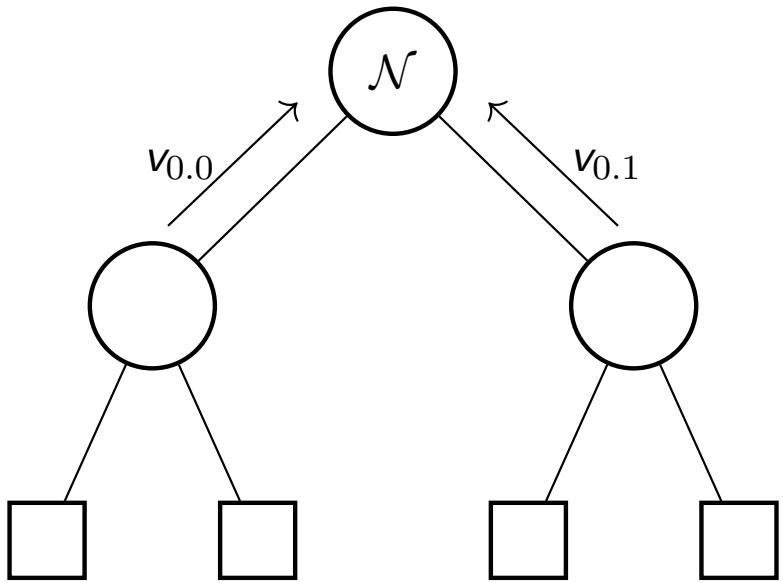
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree recursion

Recursion structure

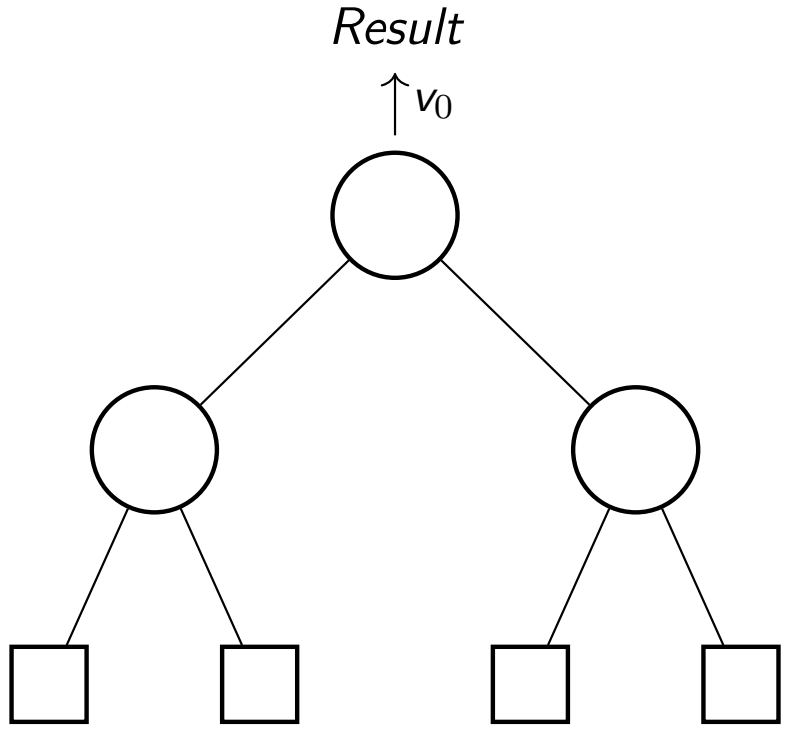
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



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

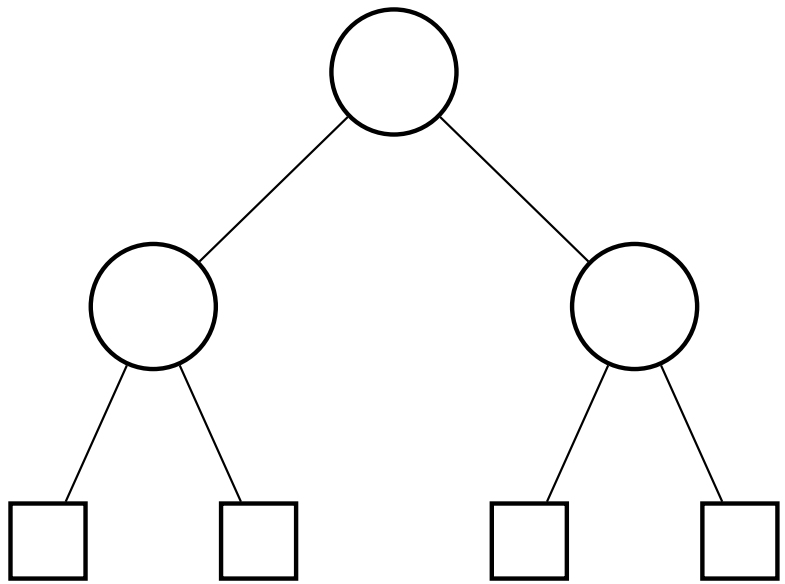
Tree construction

```
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

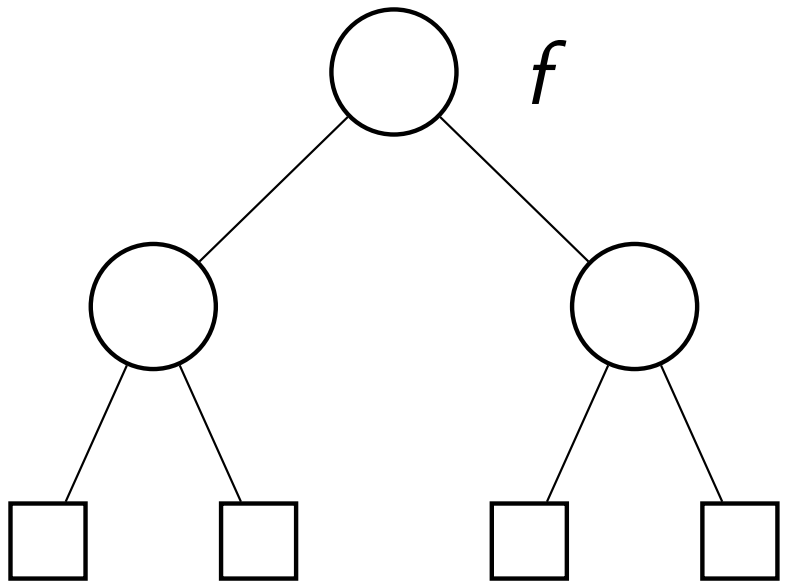
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

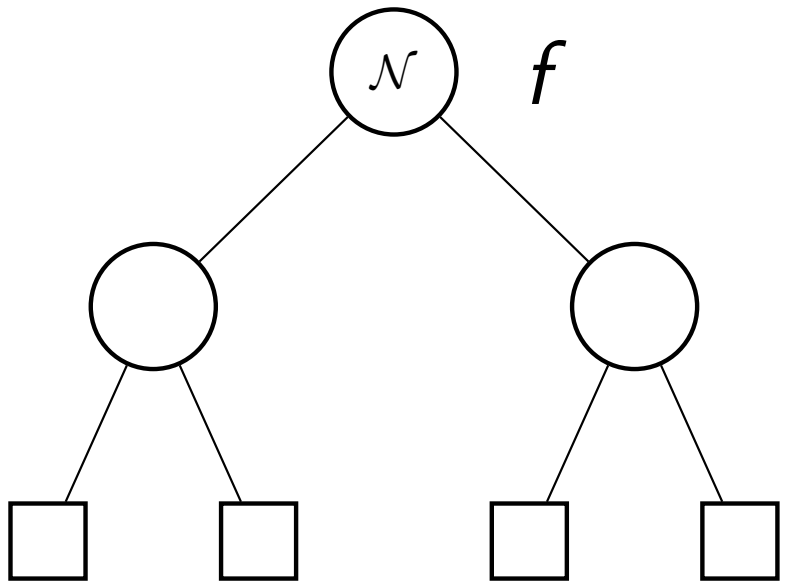
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

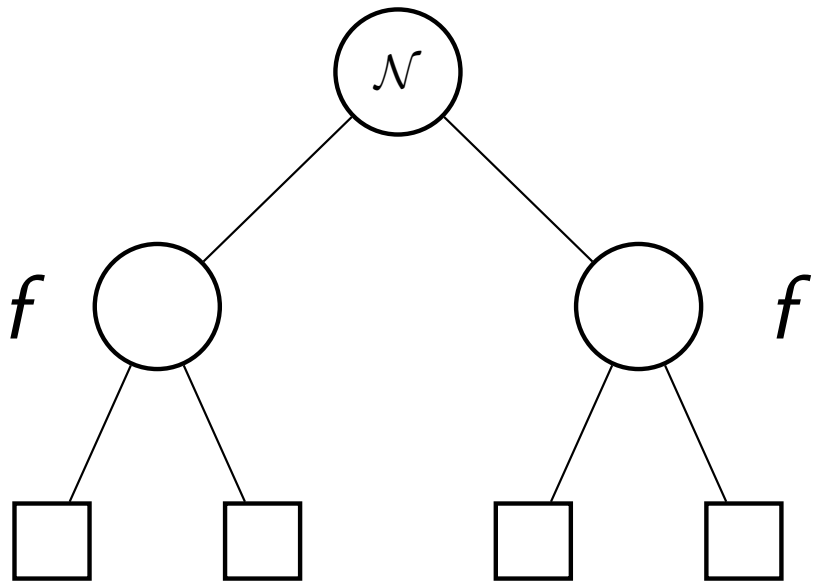
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

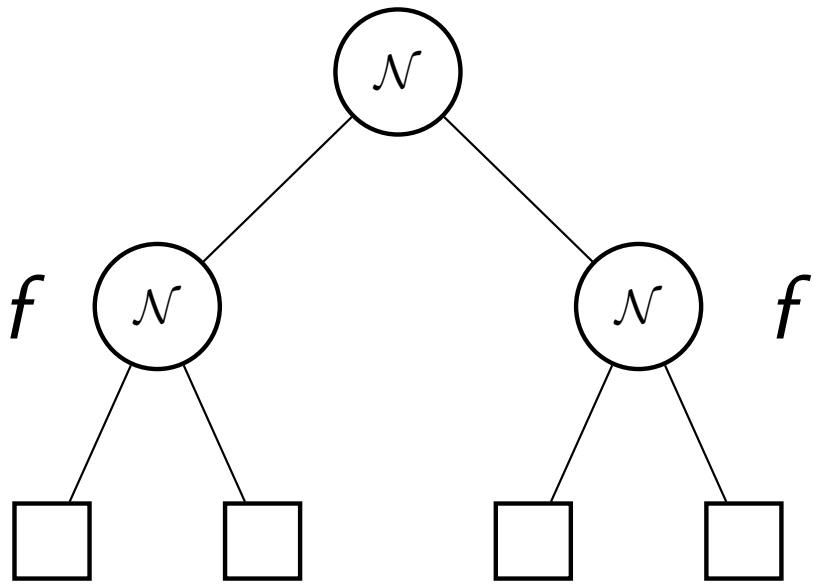
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

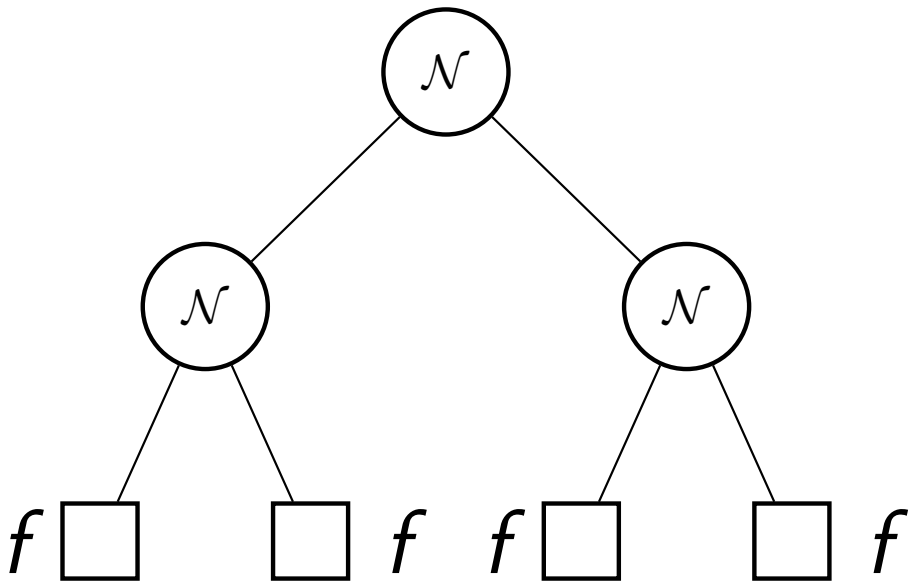
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

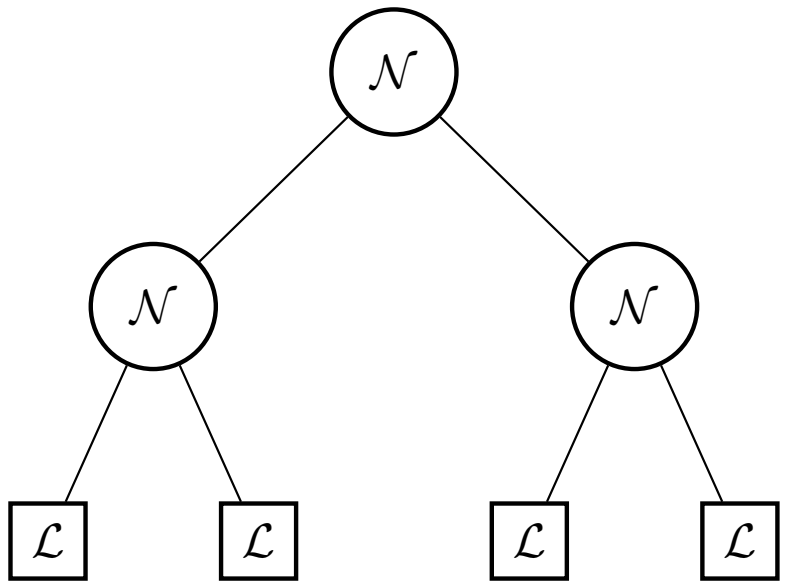
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

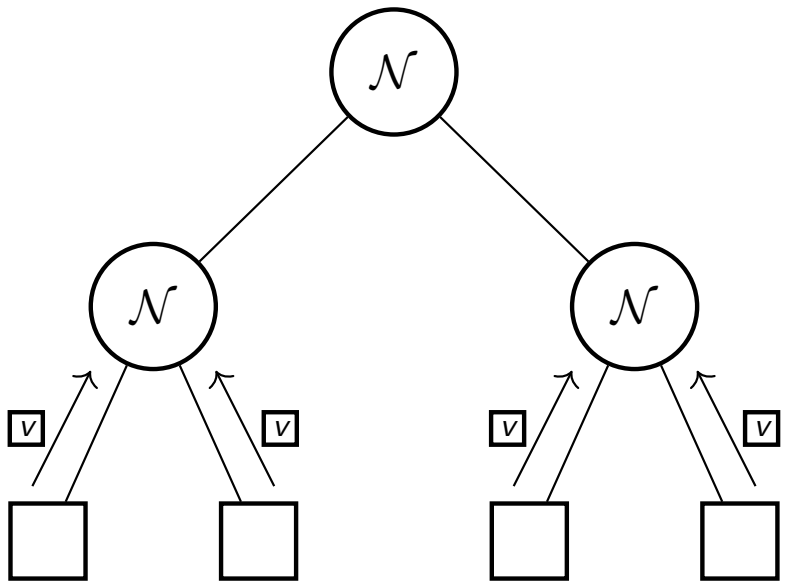
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

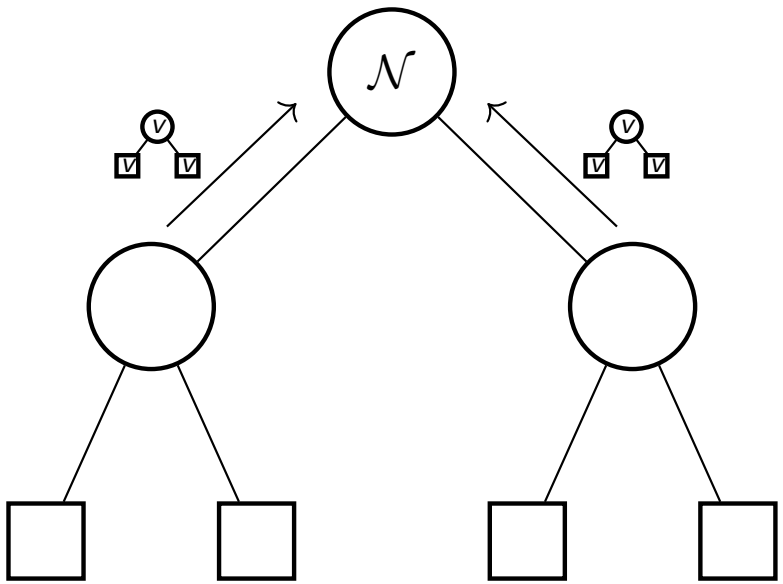
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

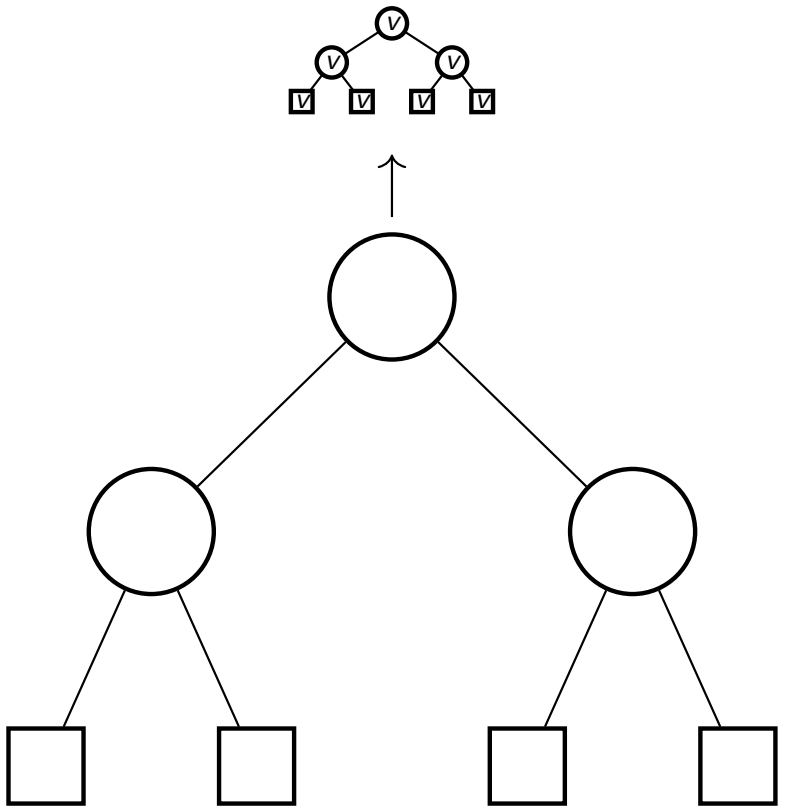
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



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
```



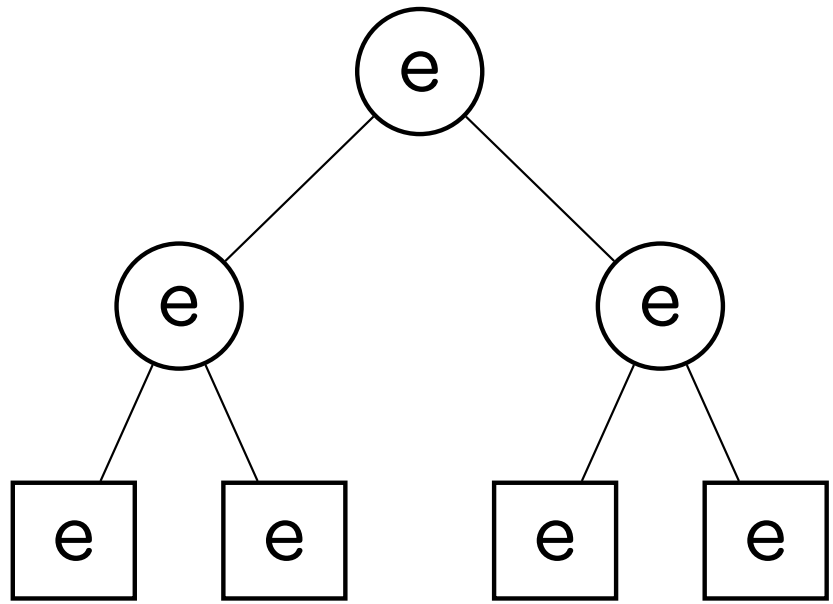
Primitives

Summary

Primitives

Summary

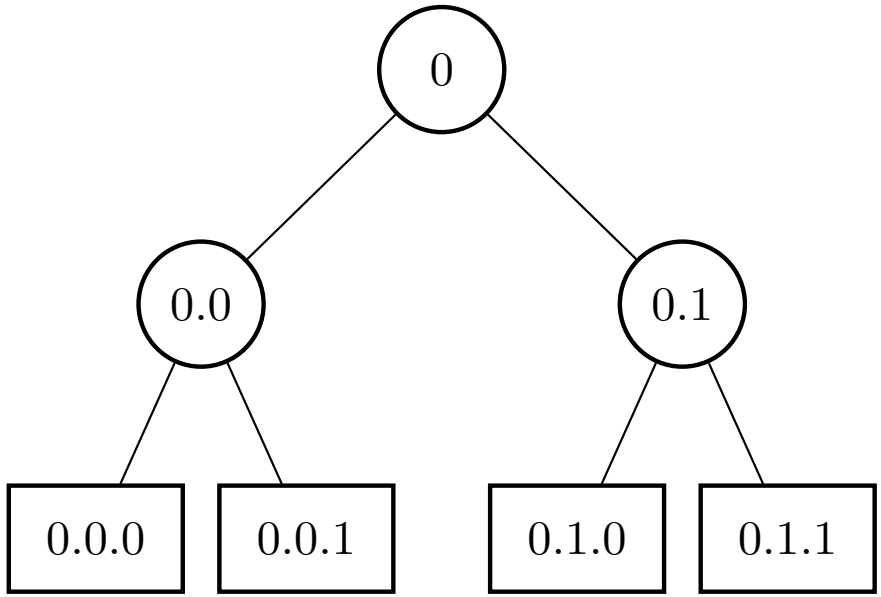
- `mktree` e



Primitives

Summary

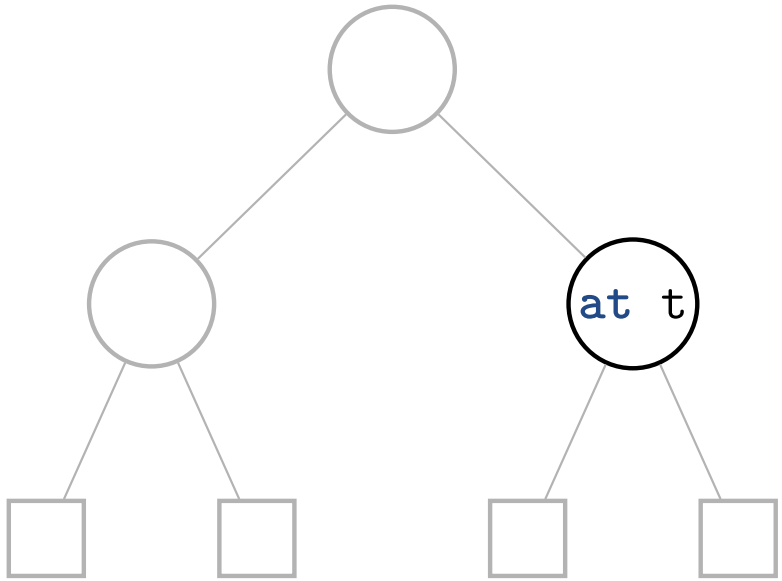
- `mktree` e
- `gid`



Primitives

Summary

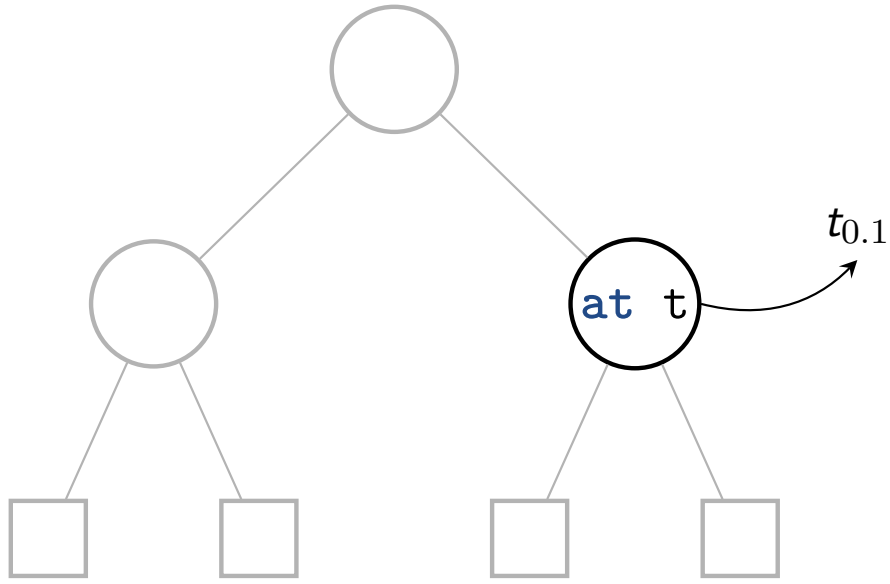
- `mktree e`
- `gid`
- `at`



Primitives

Summary

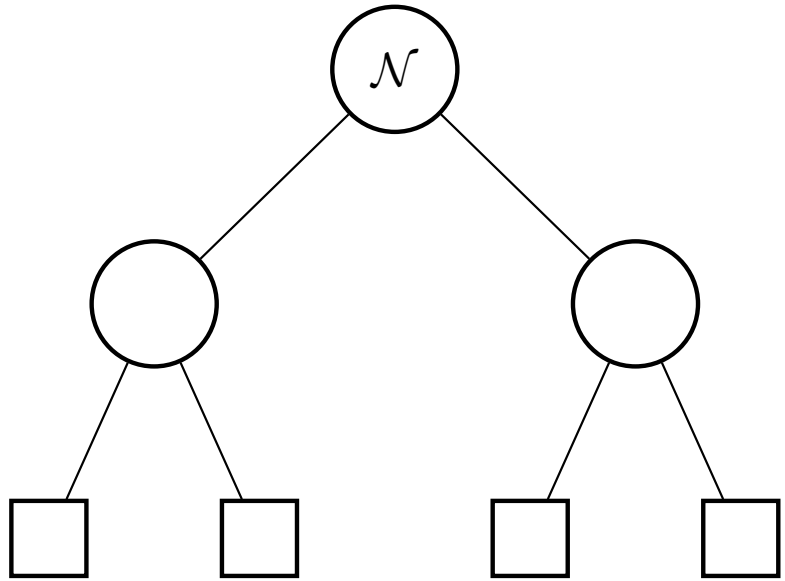
- `mktree e`
- `gid`
- `at`



Primitives

Summary

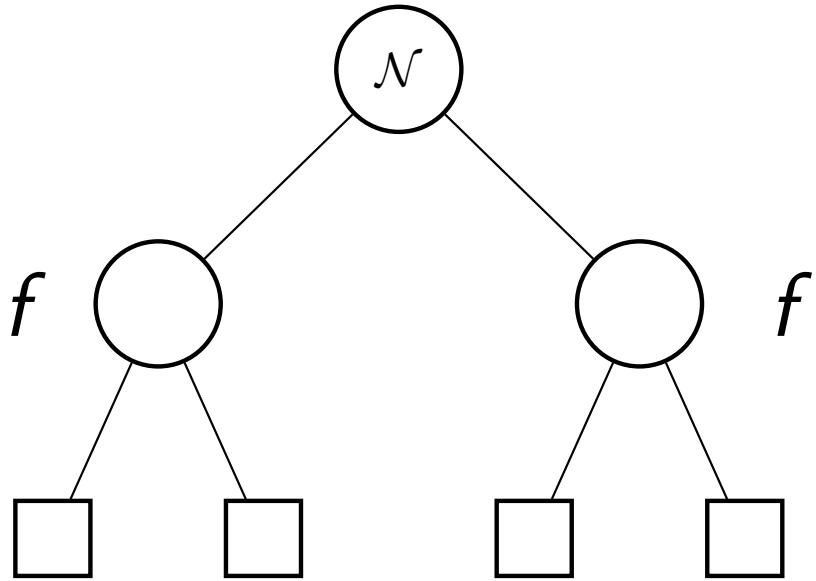
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`



Primitives

Summary

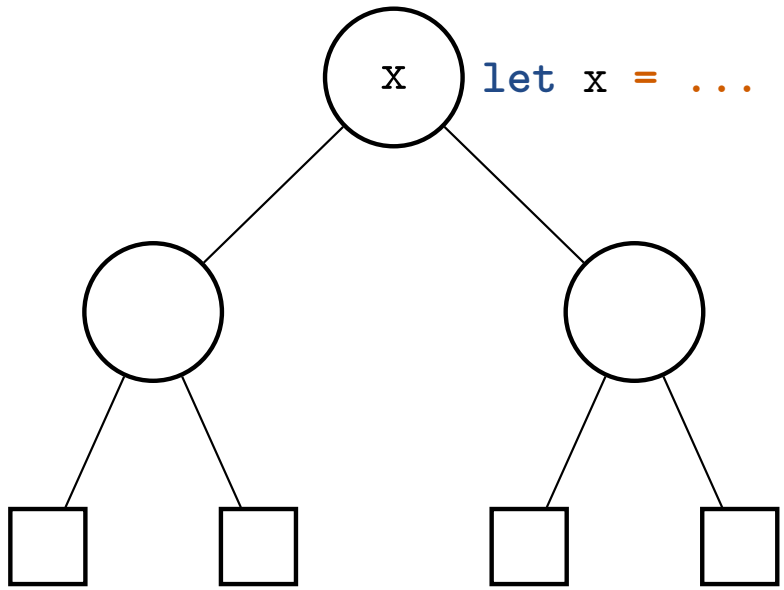
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`



Primitives

Summary

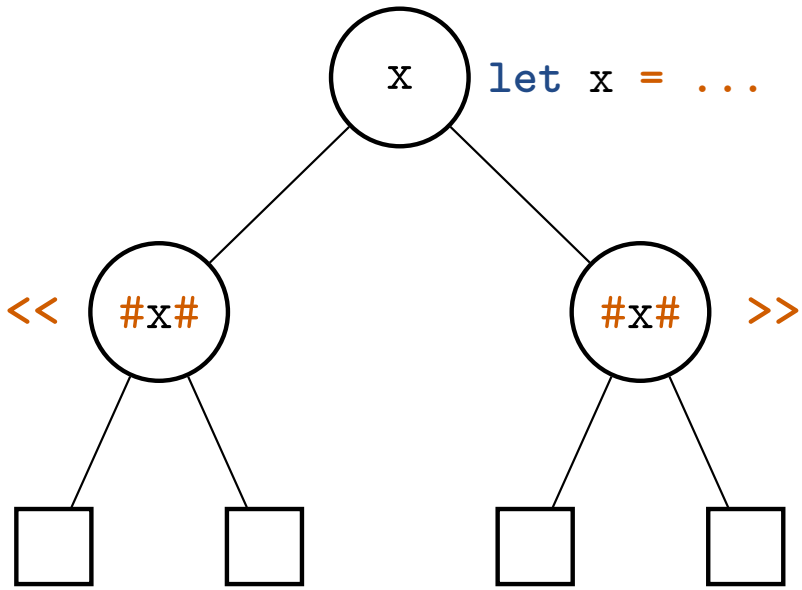
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

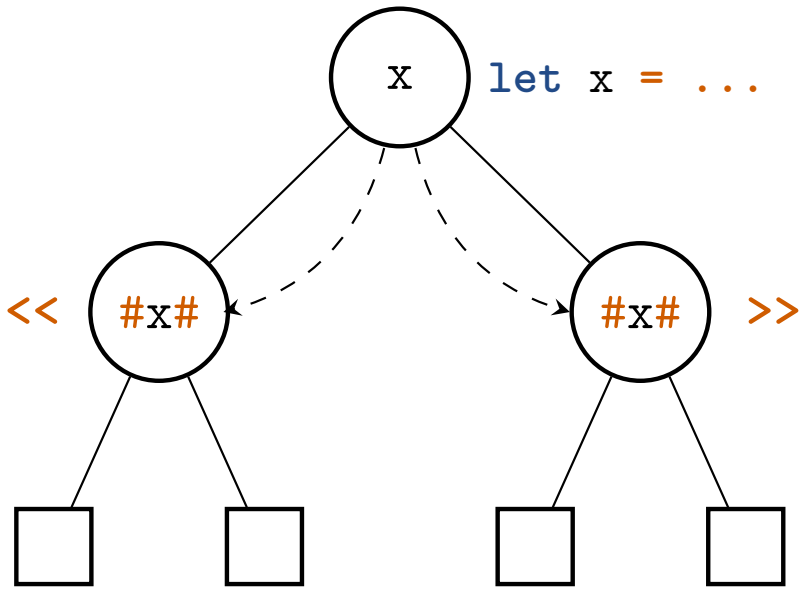
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

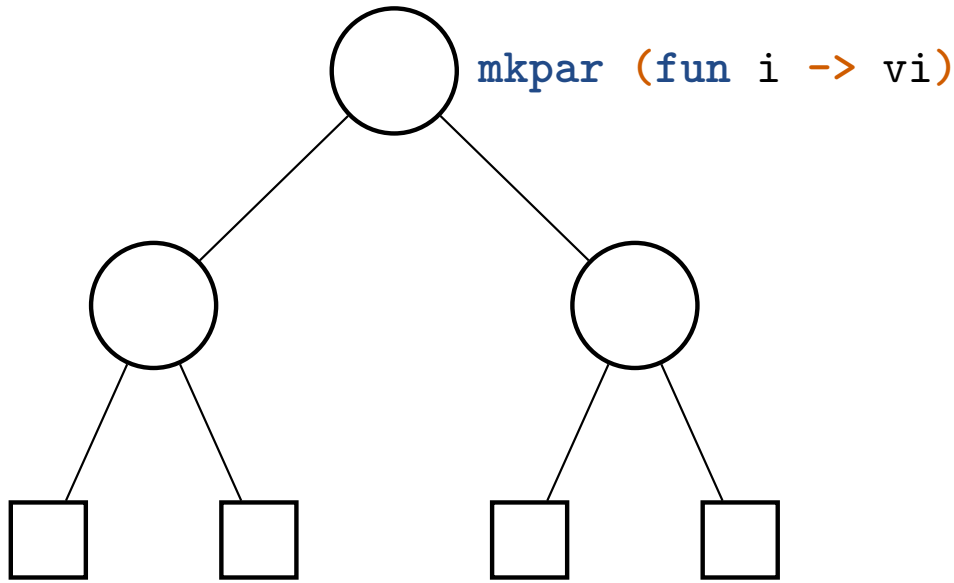
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

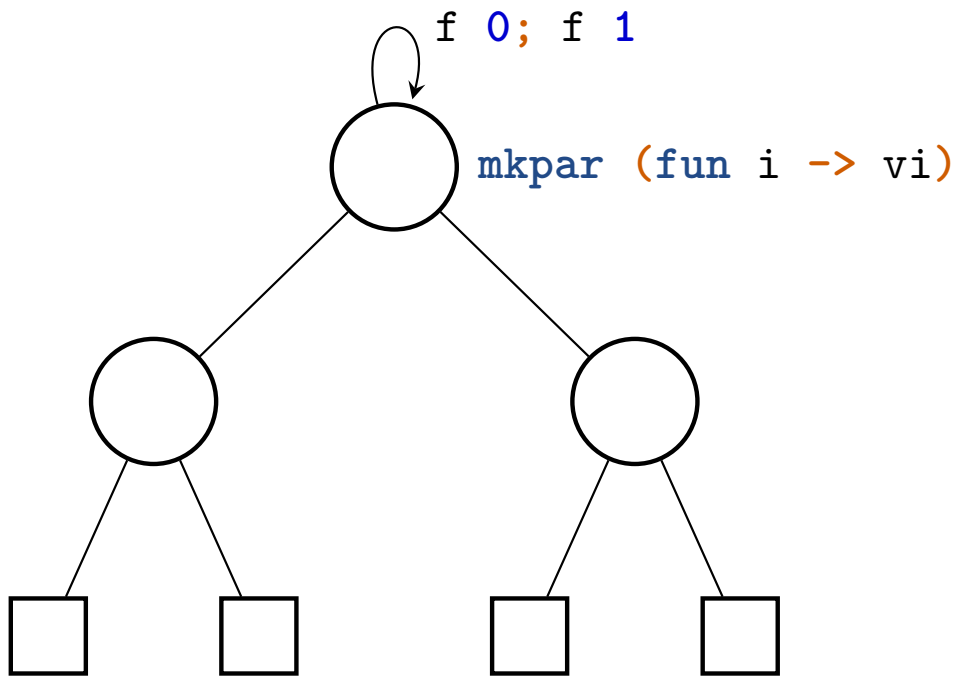
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Primitives

Summary

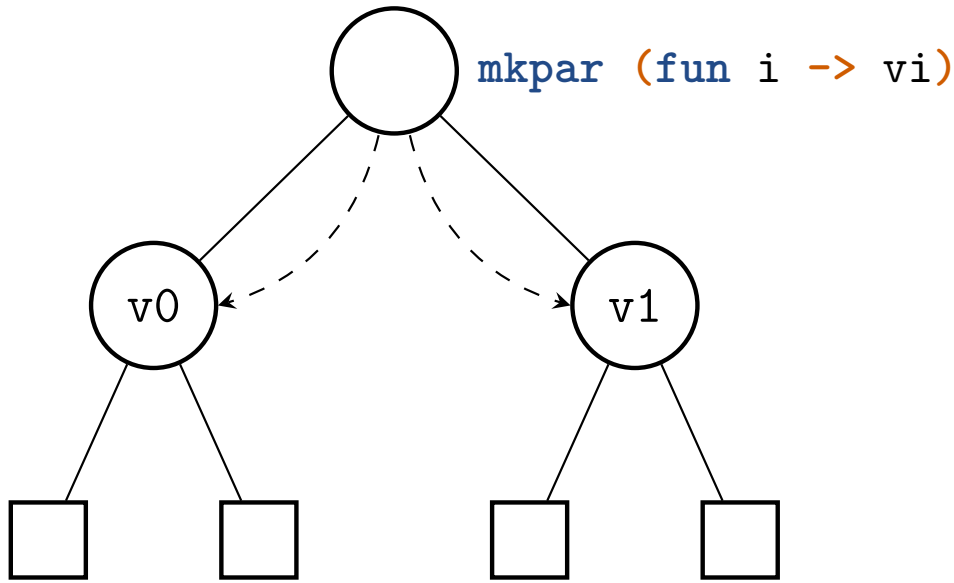
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Primitives

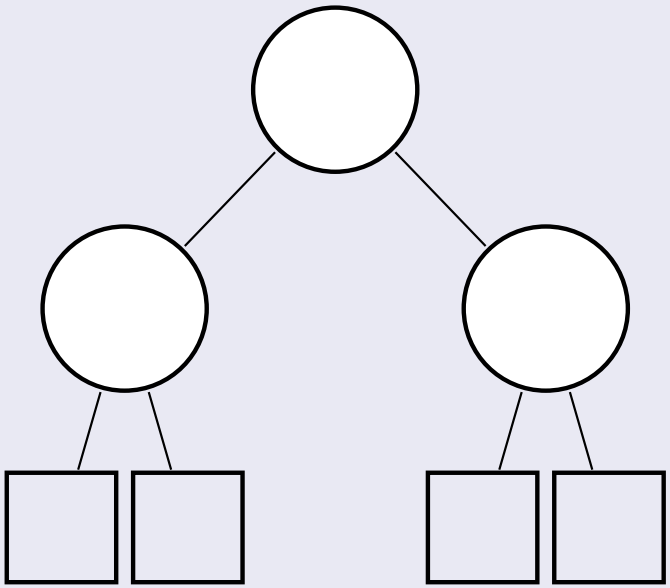
Summary

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



Code example

Keep the intermediate results of the sum

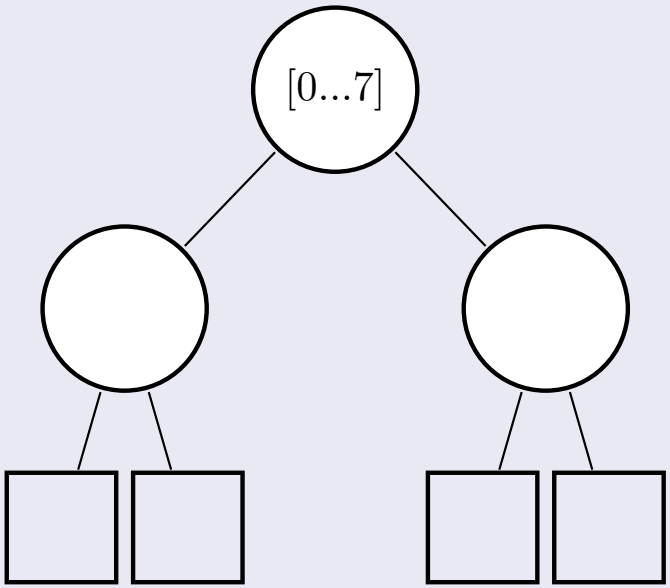


```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

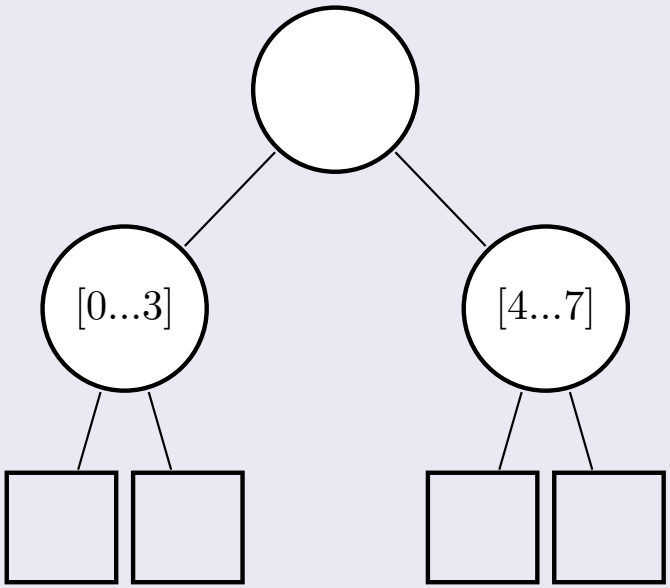



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

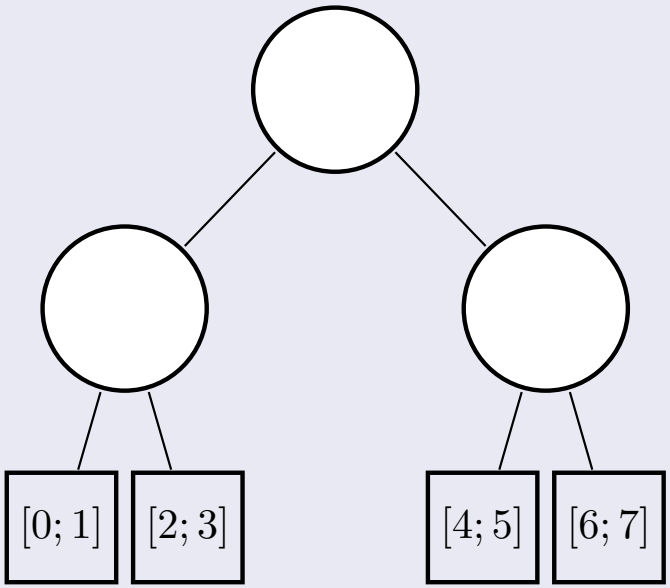



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

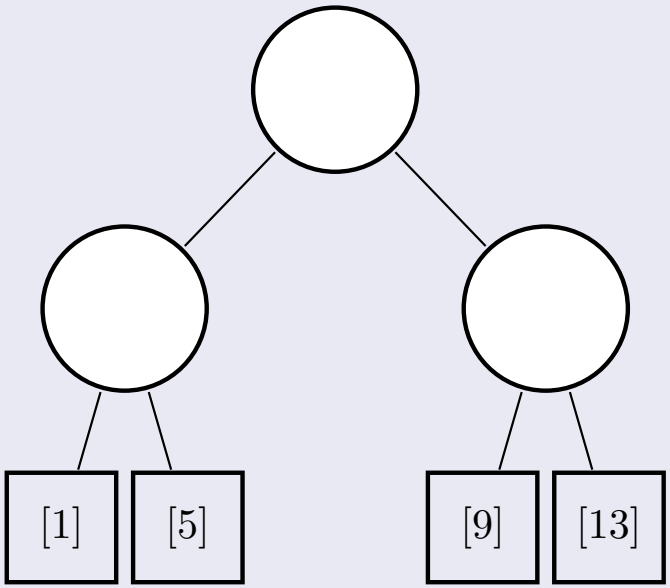



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

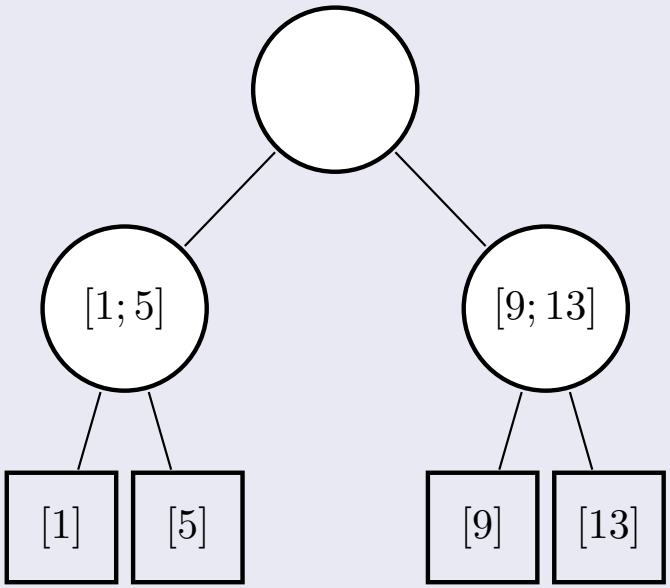



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

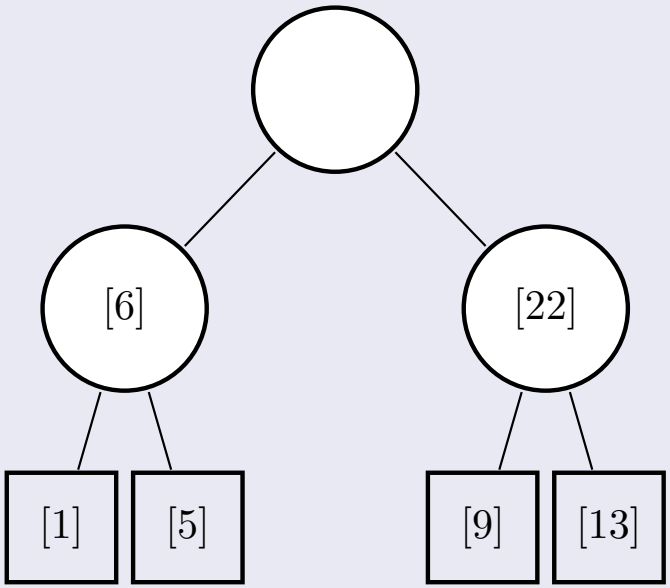



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

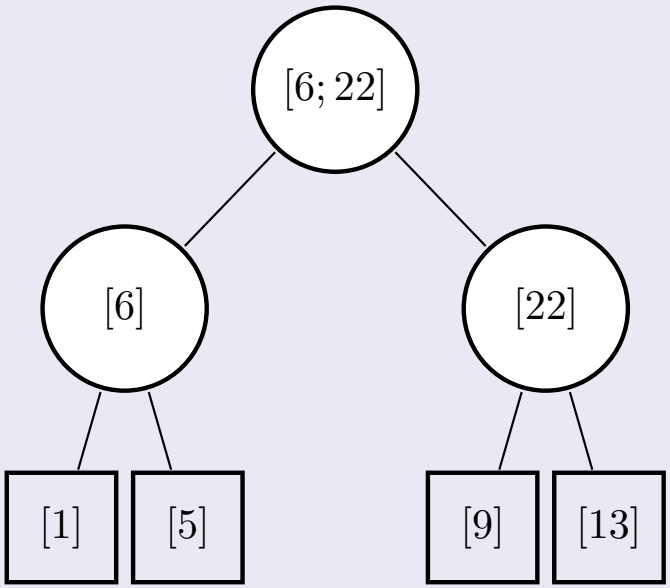



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

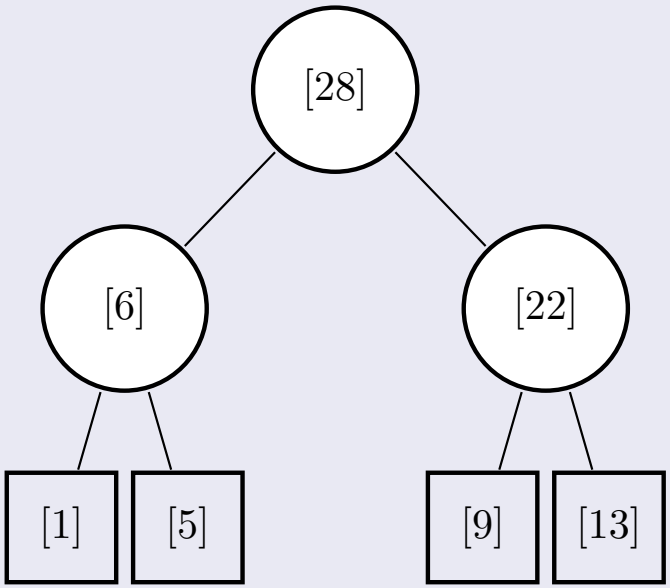



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

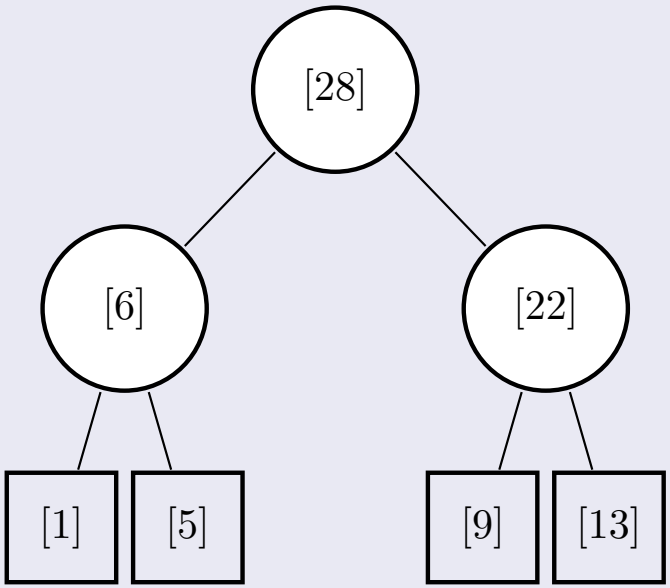



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in finally rc s
   where leaf =

      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
```

Implementation

Run on multi-core clusters using MPI.

Table of Contents

Typing system

Parallel program safety

- Replicated coherency

Replicated coherency

```
if random_bool () then
  proj v
else
  ...
```

Typing system

Parallel program safety

- Replicated coherency
- Level (memory) compatibility

Level(memory) compatibility

```
<< let multi f x = ... >>
```

Typing system

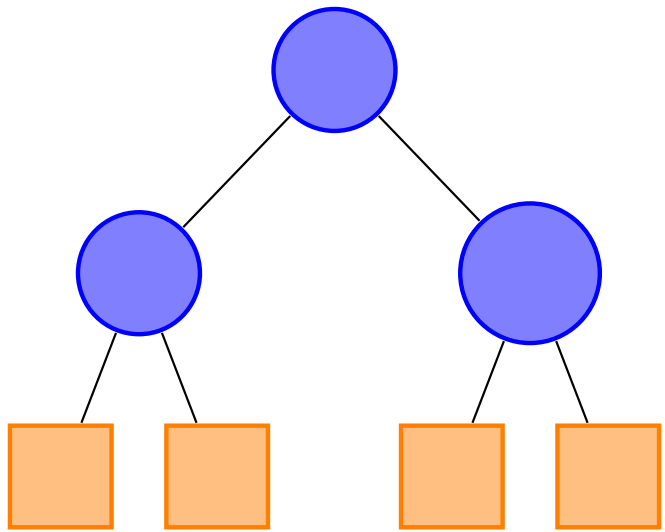
Parallel program safety

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

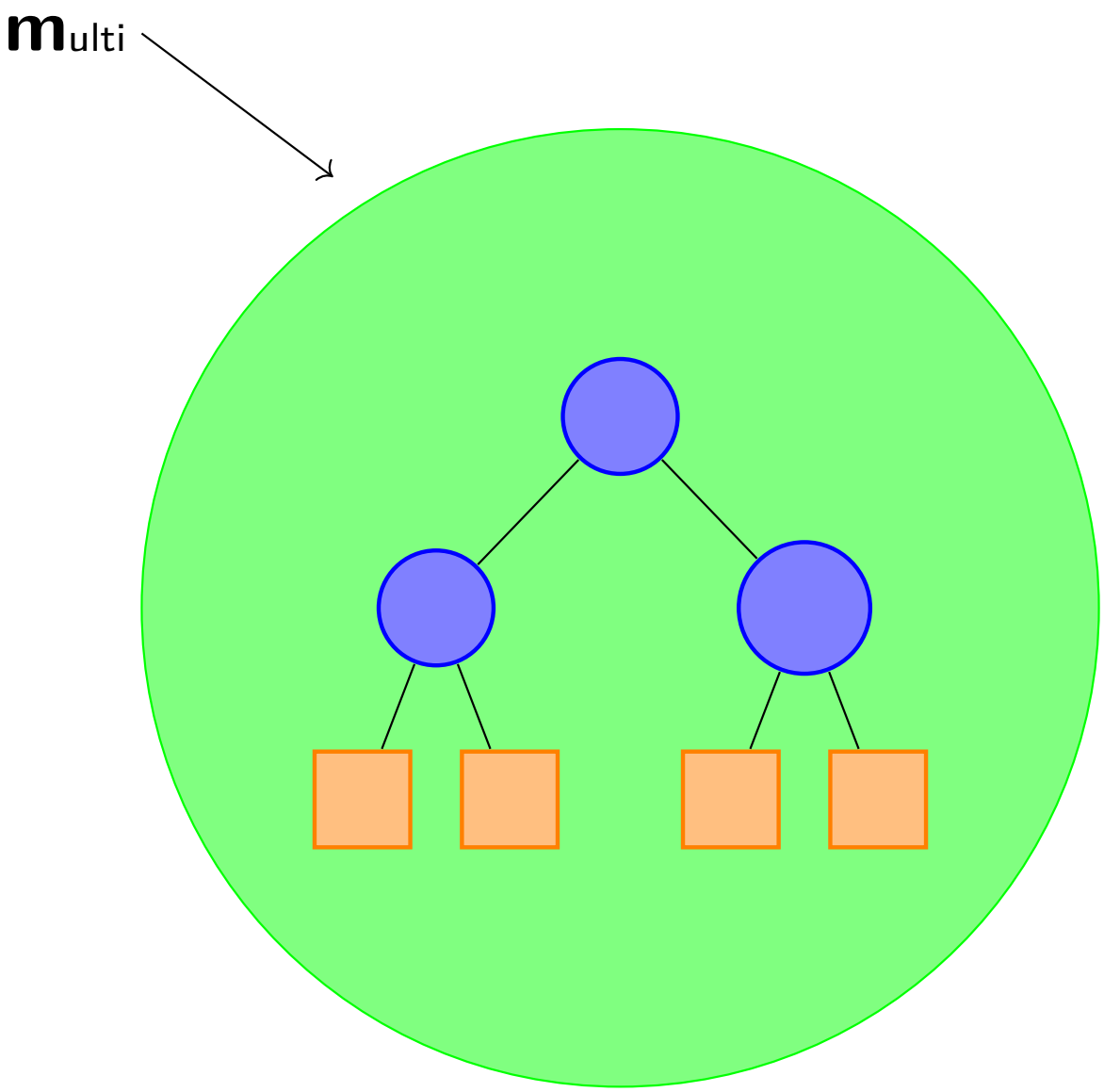
Parallel structure imbrication

```
let v = << ... >> in << v >>
```

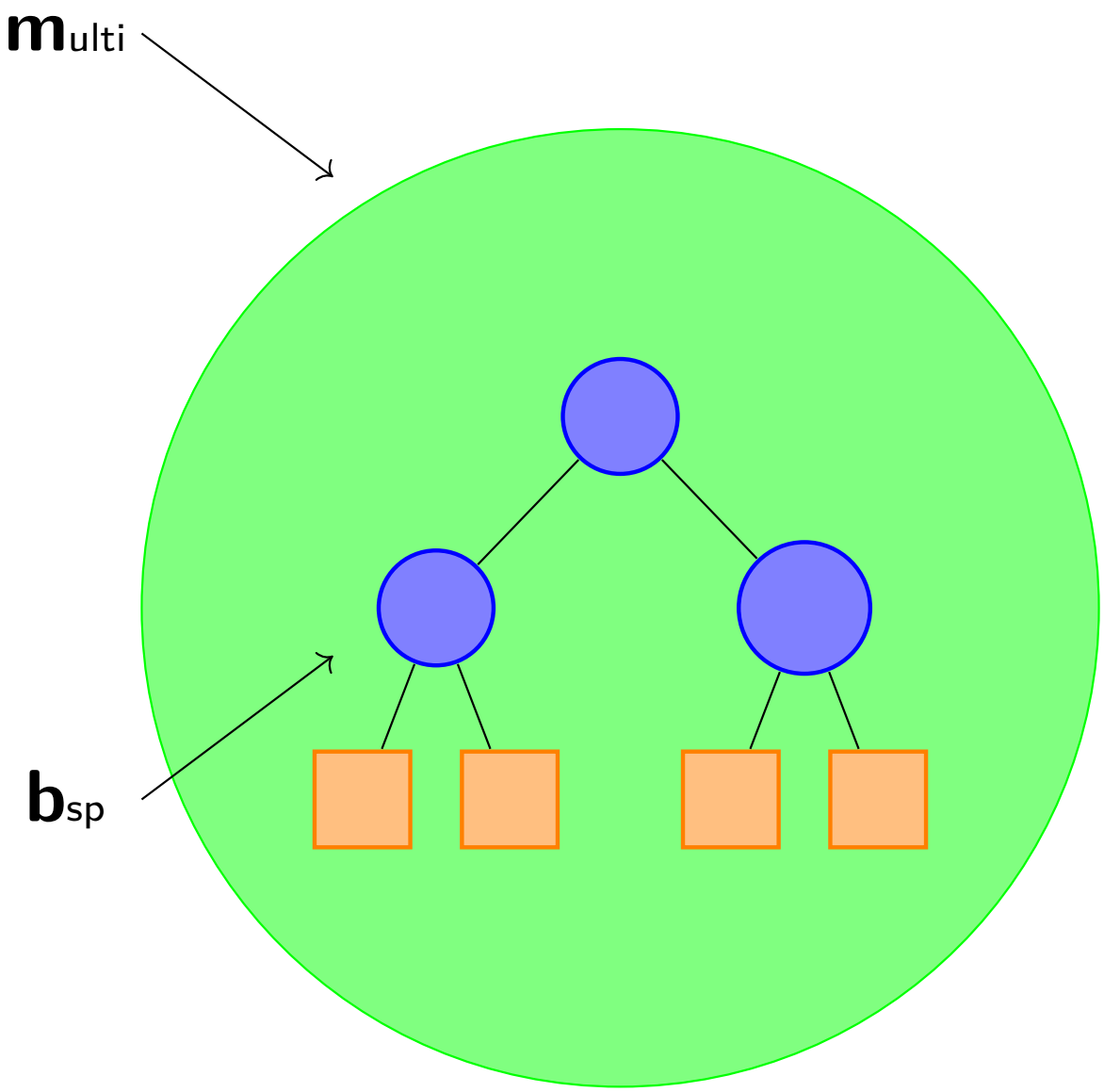
Type localities



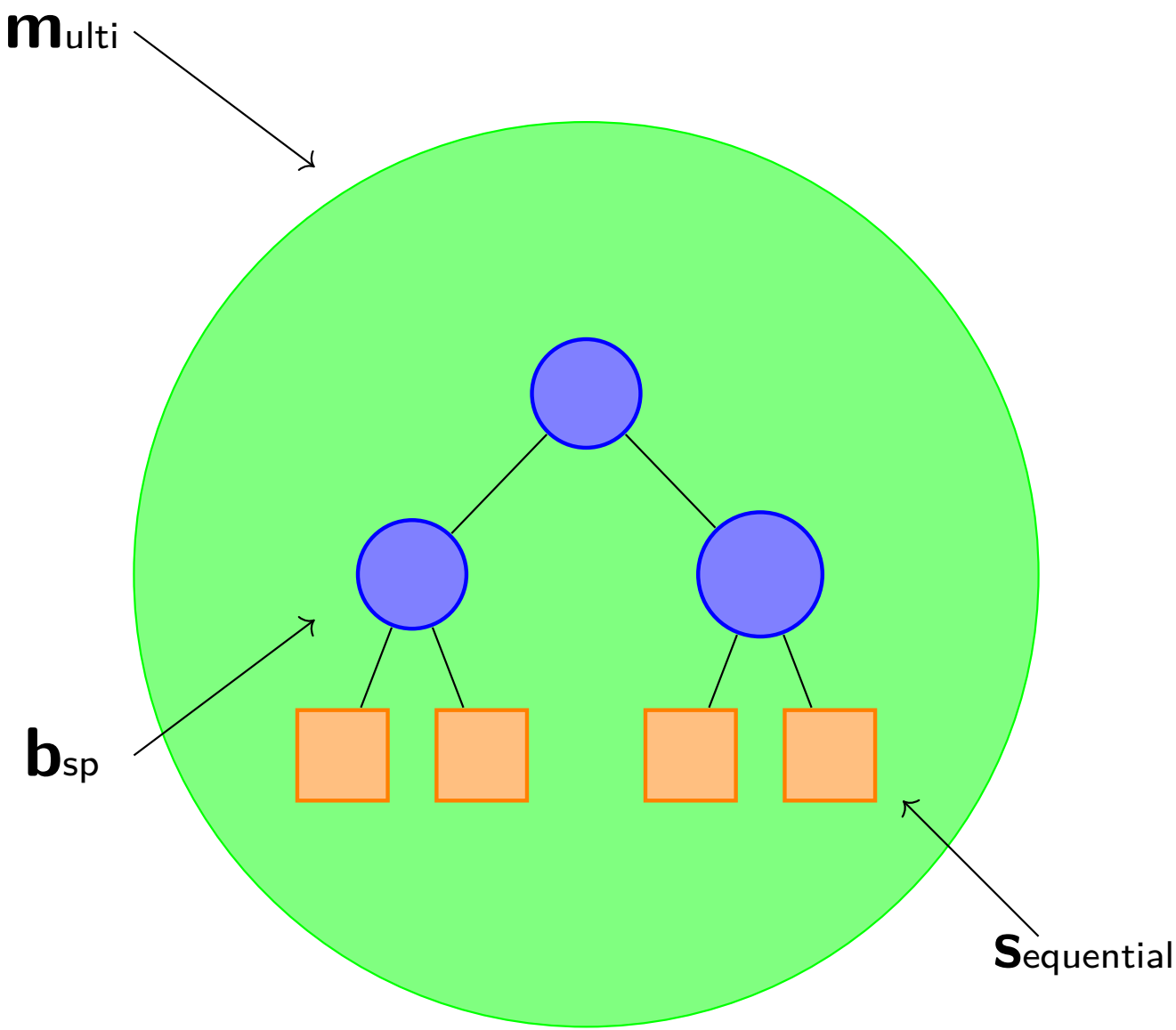
Type localities



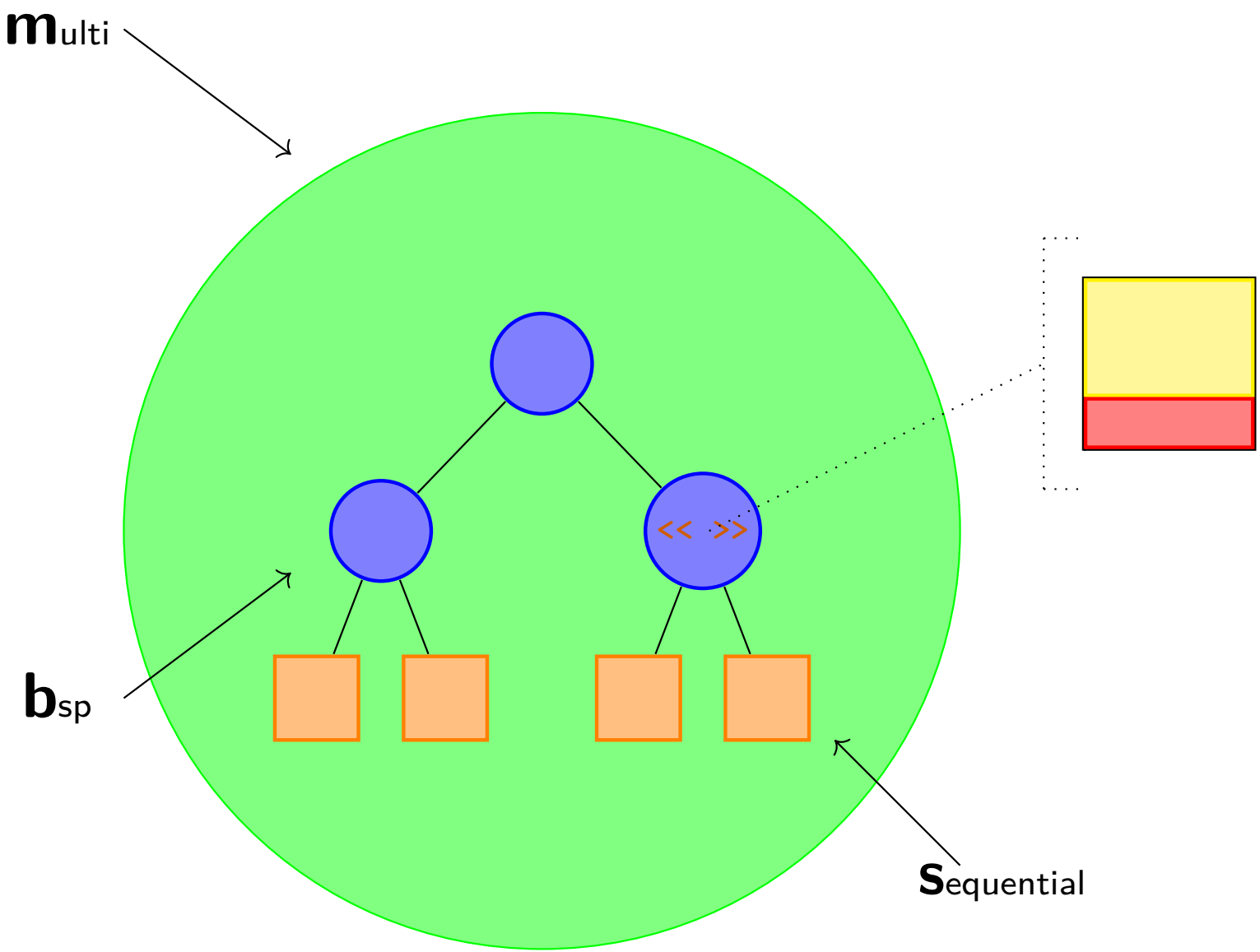
Type localities



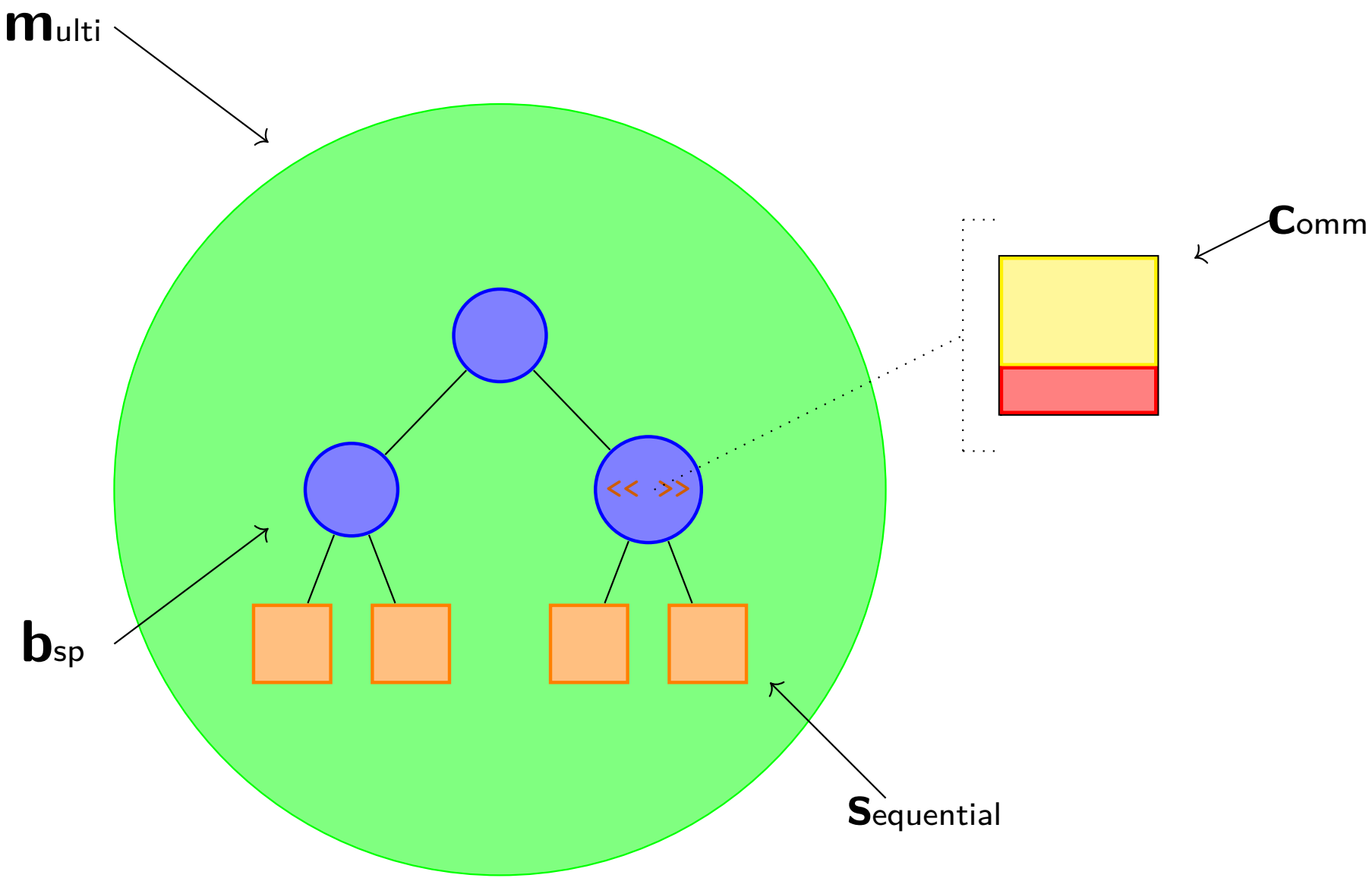
Type localities



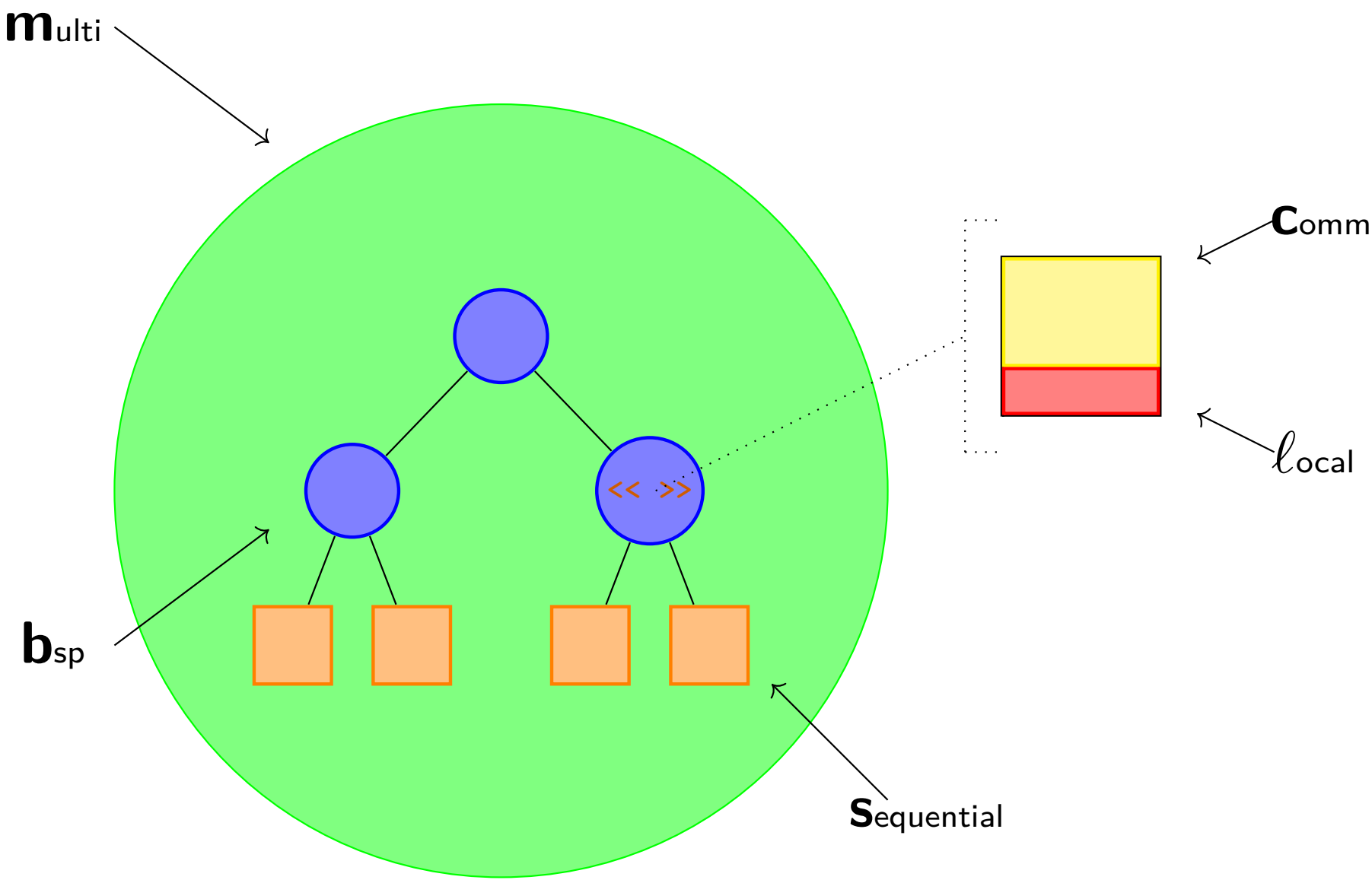
Type localities



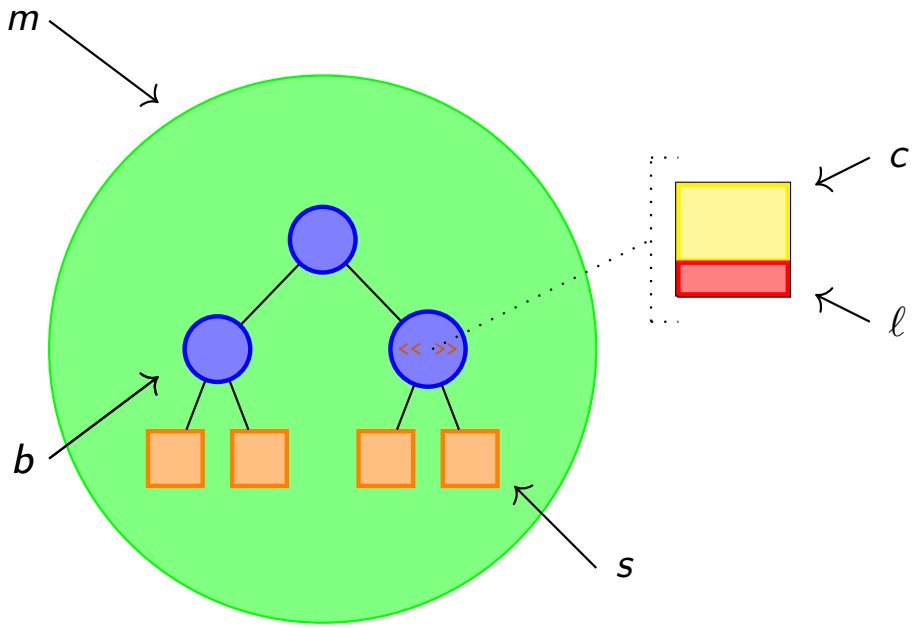
Type localities



Type localities



Type annotations



Type annotation

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

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

Type annotation

Latent effect

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

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

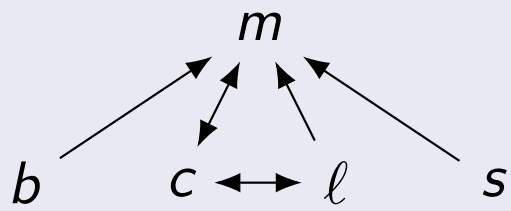
A BSP function

```
#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: $\lhd$

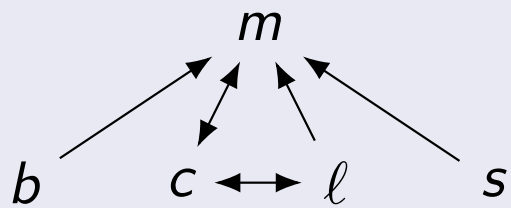
$$
\begin{aligned}
m, c &\lhd m \\
m, b &\lhd b \\
m, l, c &\lhd l \\
m, l, c &\lhd c \\
m, s &\lhd s
\end{aligned}
$$



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

Accessibility: $\lhd$

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



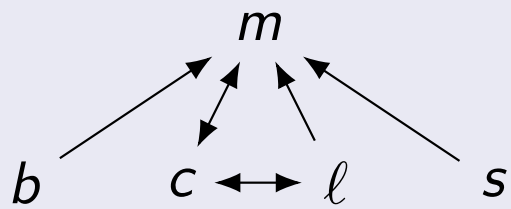
$\lambda_2 \lhd \lambda_1$: « λ_1 *can read in* λ_2 *memory.* »

Example:

$$f : ('a_{\iota_z} \xrightarrow{b} int_b)_m$$
$$f\ 1 \rightsquigarrow b \lhd m$$

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:

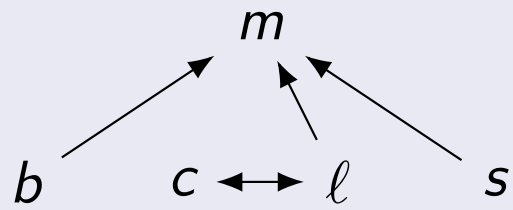
$$f : ('a_{\leadsto z} \xrightarrow{b} int_b)_m$$

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

Error

Definability: $\blacktriangleleft$

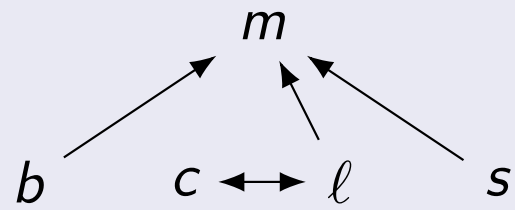
$$
\begin{array}{rcl}
s, b, m & \blacktriangleleft & m \\
b & \blacktriangleleft & b \\
l, c & \blacktriangleleft & c \\
l, c & \blacktriangleleft & l \\
s & \blacktriangleleft & s
\end{array}
$$



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

Definability: $\blacktriangleleft$

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



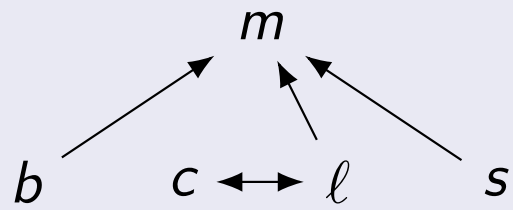
$\lambda_1 \blacktriangleleft \lambda_2$: « λ_1 *can be defined in* λ_2 *memory.* »

Example:

$$<< \textit{let multi } f\ x\ =\ \ldots >> \rightsquigarrow m \blacktriangleleft b$$

Definability: $\blacktriangleleft$

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



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

Example:

$$<< \textit{let multi } f\ x\ =\ \ldots >> \rightsquigarrow m \blacktriangleleft b$$

Error

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

$Propgt$	m	b	l	c	s
m	m	m	m	m	m
b	m	b	b	b	b
l	m	b	l	l	$\bot$
c	m	b	l	c	$\bot$
s	m	b	$\bot$	$\bot$	s

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

$Propgt$	m	b	l	c	s
m	m	m	m	m	m
b	m	b	b	b	b
l	m	b	l	l	$\bot$
c	m	b	l	c	$\bot$
s	m	b	$\bot$	$\bot$	s

Constraint generation

$[\![\ e_1\ e_2 : \tau_\Lambda/\Psi\ ]\!]^\Lambda =$

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

$Propgt$	m	b	l	c	s
m	m	m	m	m	m
b	m	b	b	b	b
l	m	b	l	l	$\bot$
c	m	b	l	c	$\bot$
s	m	b	$\bot$	$\bot$	s

Constraint generation

$$[\![\ e_1\ e_2 : \tau_\Lambda/\Psi\ ]\!]^\Lambda = [\![\ e_1 : (\alpha_\pi \xrightarrow{\varepsilon} \tau_{\pi'})_\delta/\varepsilon'\ ]\!]^\Lambda$$

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

Propgt	*m*	*b*	*l*	*c*	*s*
m	*m*	*m*	*m*	*m*	*m*
b	*m*	*b*	*b*	*b*	*b*
l	*m*	*b*	*l*	*l*	$\perp$
c	*m*	*b*	*l*	*c*	$\perp$
s	*m*	*b*	$\perp$	$\perp$	*s*

Constraint generation

$$[\![\ e_1\ e_2 : \tau_\Lambda/\Psi\ ]\!]^\Lambda = [\![\ e_1 : (\alpha_\pi \xrightarrow{\varepsilon} \tau_{\pi'})_\delta/\varepsilon'\ ]\!]^\Lambda \wedge [\![\ e_2 : \alpha_{\pi''}/\varepsilon''\ ]\!]^\Lambda$$

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

Propgt	*m*	*b*	*l*	*c*	*s*
m	*m*	*m*	*m*	*m*	*m*
b	*m*	*b*	*b*	*b*	*b*
l	*m*	*b*	*l*	*l*	$\perp$
c	*m*	*b*	*l*	*c*	$\perp$
s	*m*	*b*	$\perp$	$\perp$	*s*

Constraint generation

$$[\![\ e_1\ e_2 : \tau_\Lambda / \Psi\ ]\!]^\Lambda = [\![\ e_1 : (\alpha_\pi \xrightarrow{\varepsilon} \tau_{\pi'})_\delta / \varepsilon'\ ]\!]^\Lambda \wedge [\![\ e_2 : \alpha_{\pi''} / \varepsilon''\ ]\!]^\Lambda \wedge \pi'' \lhd \pi$$

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

Propgt	*m*	*b*	*l*	*c*	*s*
m	*m*	*m*	*m*	*m*	*m*
b	*m*	*b*	*b*	*b*	*b*
l	*m*	*b*	*l*	*l*	$\bot$
c	*m*	*b*	*l*	*c*	$\bot$
s	*m*	*b*	$\bot$	$\bot$	*s*

Constraint generation

$$[\![\ e_1\ e_2 : \tau_\Lambda/\Psi\ ]\!]^\Lambda = [\![\ e_1 : (\alpha_\pi \xrightarrow{\varepsilon} \tau_{\pi'})_\delta/\varepsilon'\ ]\!]^\Lambda \wedge [\![\ e_2 : \alpha_{\pi''}/\varepsilon''\ ]\!]^\Lambda \wedge \pi'' \lhd \pi \ \ \wedge \varepsilon \lhd \Lambda$$

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

Propgt	*m*	*b*	*I*	*c*	*s*
m	*m*	*m*	*m*	*m*	*m*
b	*m*	*b*	*b*	*b*	*b*
I	*m*	*b*	*I*	*I*	$\bot$
c	*m*	*b*	*I*	*c*	$\bot$
s	*m*	*b*	$\bot$	$\bot$	*s*

Constraint generation

$$[\![\ e_1\ e_2 : \tau_\Lambda / \Psi\ ]\!]^\Lambda = [\![\ e_1 : (\alpha_\pi \xrightarrow{\varepsilon} \tau_{\pi'})_\delta / \varepsilon'\ ]\!]^\Lambda \wedge [\![\ e_2 : \alpha_{\pi''} / \varepsilon''\ ]\!]^\Lambda \wedge \pi'' \lhd \pi \ \wedge \varepsilon \lhd \Lambda \ \wedge \varepsilon \blacktriangleleft \Lambda$$

Propagation: $Propgt(\varepsilon, \varepsilon')$

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

$Propgt$	m	b	l	c	s
m	m	m	m	m	m
b	m	b	b	b	b
l	m	b	l	l	$\perp$
c	m	b	l	c	$\perp$
s	m	b	$\perp$	$\perp$	s

Constraint generation

$$[\![\ e_1\ e_2 : \tau_\Lambda/\Psi\ ]\!]^\Lambda = [\![\ e_1 : (\alpha_\pi \xrightarrow{\varepsilon} \tau_{\pi'})_\delta/\varepsilon'\ ]\!]^\Lambda \wedge [\![\ e_2 : \alpha_{\pi''}/\varepsilon''\ ]\!]^\Lambda \wedge \pi'' \lhd \pi \ \wedge \varepsilon \lhd \Lambda \ \wedge \varepsilon \blacktriangleleft \Lambda \ \wedge \Psi = Propgt(\varepsilon, \varepsilon', \varepsilon'')$$

Serialisation: $Seria_{\Lambda}(\tau_{\pi})$

Is it safe to communicate τ_{π} to locality Λ ?

$$
\begin{array}{lcl}
Seria_{\alpha}(\tau_{\pi}) & = & \begin{cases} \tau_{\pi}, & \textit{if } \pi \lhd \alpha \end{cases} \\
Seria_{\alpha}(\texttt{Base}_{\pi}) & = & \texttt{Base}_{\pi} \ \textit{if } \texttt{Base} = \texttt{int}, \texttt{Bool}, \ldots \\
Seria_{\alpha}(\texttt{Base}_{\pi}) & = & \texttt{Fail} \ \textit{if } \texttt{Base} = \texttt{i/o}, \ldots \\
Seria_{\alpha}(\tau_{\pi}\ \texttt{par}_{b}) & = & \texttt{Fail} \\
Seria_{\alpha}(\texttt{tree}_{\pi}) & = & \texttt{Fail} \\
Seria_{\alpha}(\_ \xrightarrow{l} \_) & = & \texttt{Fail} \\
\ldots & & \\
Seria_{\alpha}((\tau^{1}_{\pi} \xrightarrow{\varepsilon} \tau^{2}_{\pi'})_{\delta}) & = & \begin{cases} (\tau'^{1}_{\pi} \xrightarrow{\varepsilon} \tau'^{2}_{\pi'})_{\delta}, & \textit{if } \varepsilon \# m, b, s, l \\ & \textit{and } \tau'^{1}_{\pi} = Seria_{\alpha}(\tau^{1}_{\pi}) \\ & \textit{and } \tau'^{2}_{\pi'} = Seria_{\alpha}(\tau^{2}_{\pi'}) \end{cases}
\end{array}
$$

Table of Contents

Conclusion

MULTI-ML

Conclusion

MULTI-ML

- Recursive multi-functions

Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes

Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Small number of primitives and little syntax extension

Conclusion

MULTI-ML

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

Conclusion

MULTI-ML

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

Conclusion

MULTI-ML

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

Current/Future work

- Full implementation (record, tuples, ...)
- Variants
- Modules and other OCAML features
- Error tracking

Merci !

Any questions ?