A type system to avoid runtime errors for Multi-ML

Frédéric Gava[1] & Victor Allombert[2] & Julien Tesson[2]

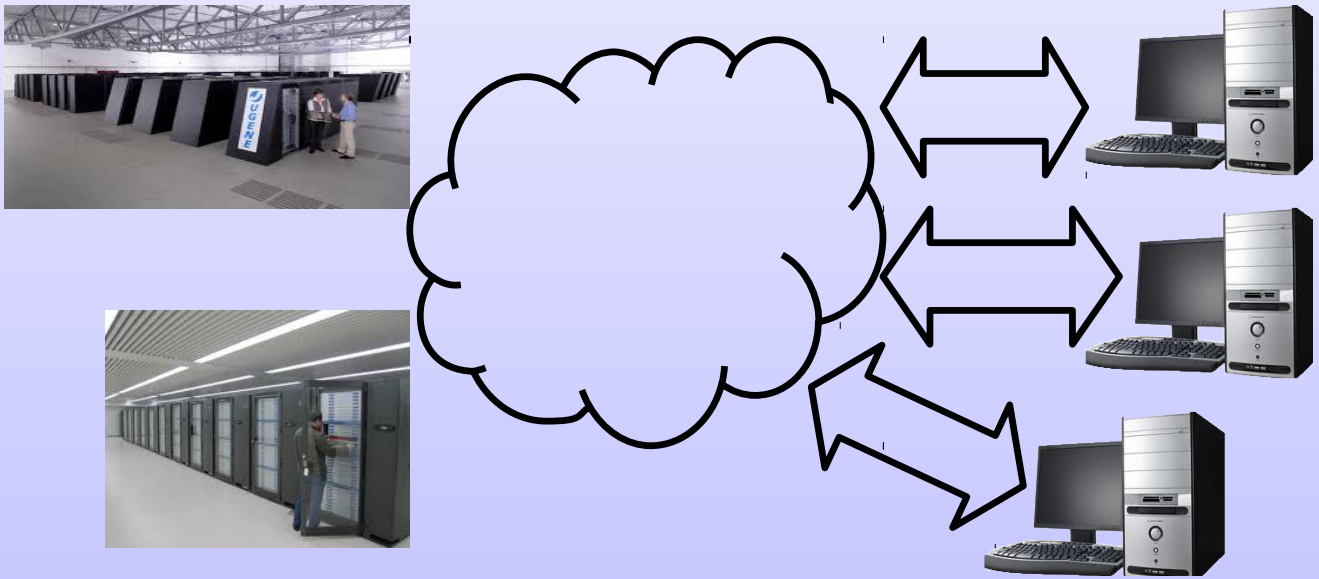
[1] **L**aboratory of **A**lgorithms, **C**omplexity and **L**ogic (LACL)
University of Paris-East, France

[2] Former LACL members

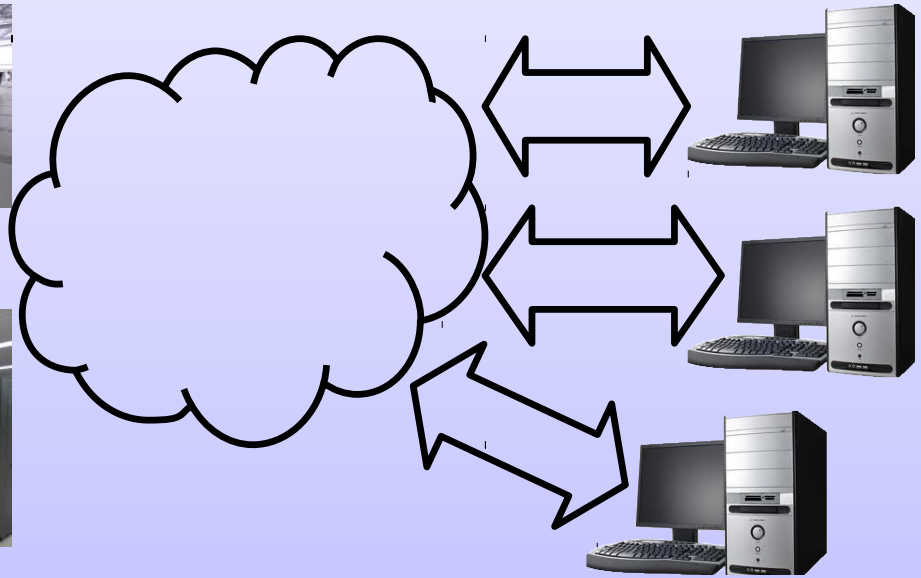
Outline

Outline

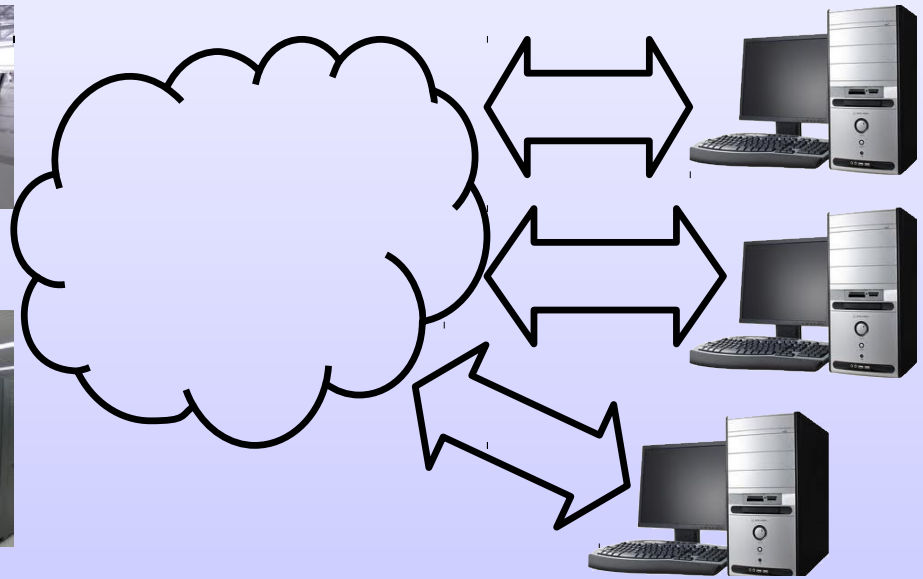
Q: Cloud/Grid computing?



Q: Cloud/Grid computing?



Q: Cloud/Grid computing?



Bugs and crashing of HPC programs

- Portability, scalability ⇒ software-hardware bridging model
- Safety, cost ⇒ Structured parallelism ⇒ (Multi-)BSP

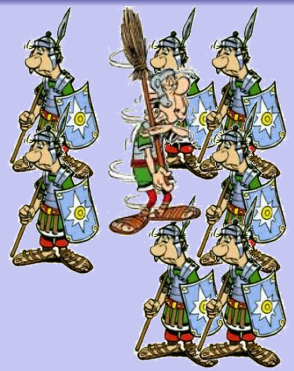
Q: why structured parallelism?

Debugging and verification

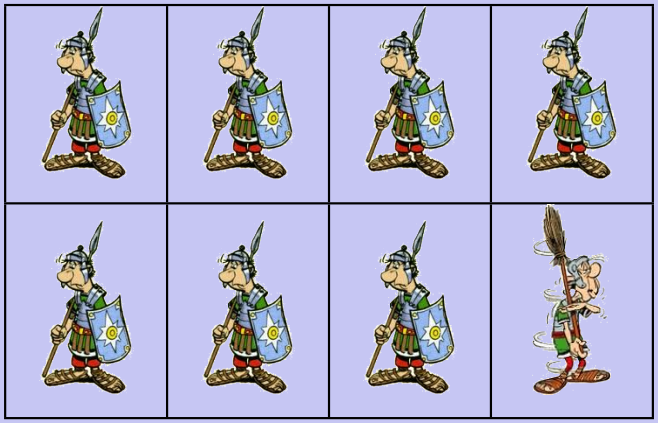
Reasoning about cost

Q: why structured parallelism?

Debugging and verification



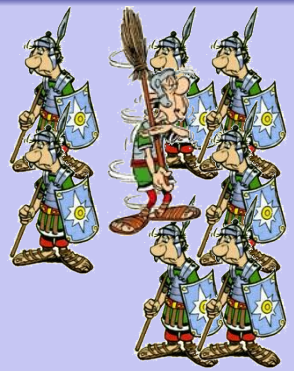
⇒



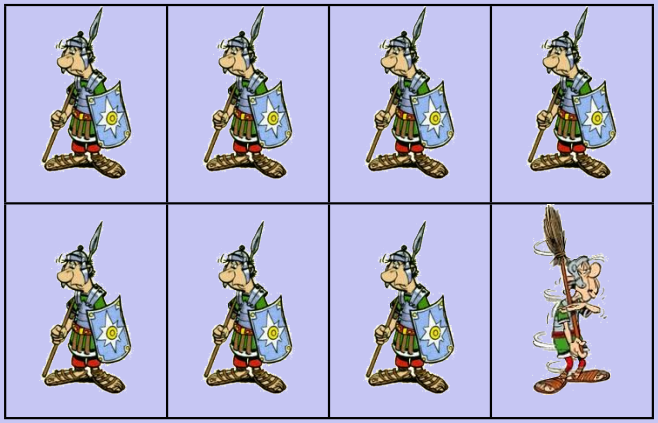
Reasoning about cost

Q: why structured parallelism?

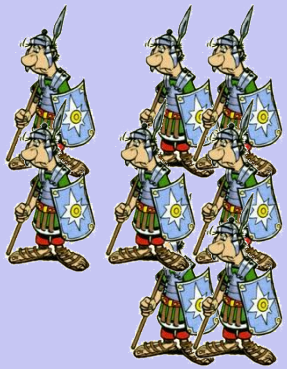
Debugging and verification



⇒



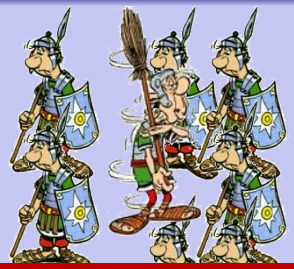
Reasoning about cost



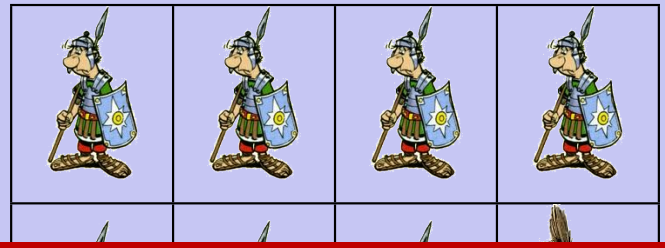
⇒

Q: why structured parallelism?

Debugging and verification



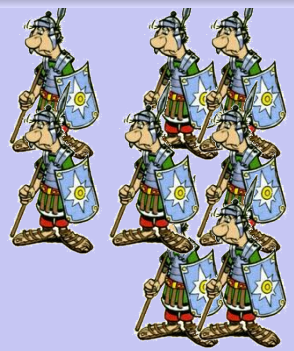
⇒



Considered solutions

"*Send-receive considered harmful*" (Sergei GORLATCH)

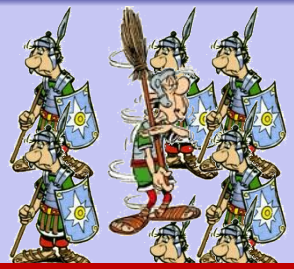
1. (Multi-)BSP extension of a functional language
2. Types and Semantics

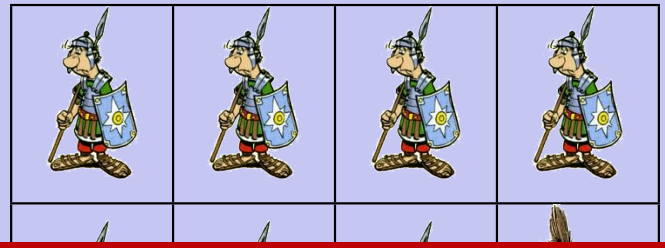


⇒

Q: why structured parallelism?

Debugging and verification

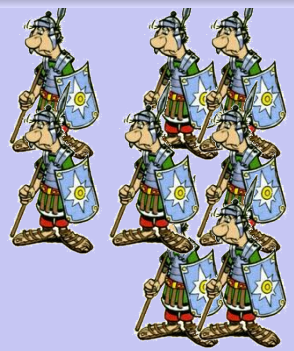




Considered solutions

"*Send-receive considered harmful*" (Sergei GORLATCH)

1. (Multi-)BSP extension of a functional language
2. Types and Semantics



Q: What is OCaml?

A general-purpose (but functional spirit) programming language

- Rich static type system
- Many features: lambda, objects, modules, GADT, *etc.*
- Industrial strength (many tools and libraries)

Some short examples

```
# let compo f g x = g (f x)
# type 'a list = Null | Node of 'a * 'a list
# let rec length l = match l with
    | Null → 0
    | Node (data,next) → 1+(length next);;
```

Q: What is OCaml?

A general-purpose (but functional spirit) programming language

- Rich static type system
- Many features: lambda, objects, modules, GADT, *etc.*

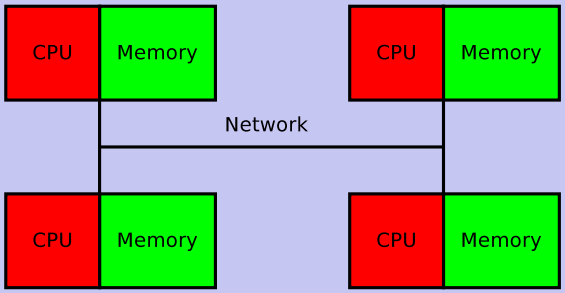
And now...

1. Multi-BSP (hierachical version of BSP)
2. Multi-BSP extention for OCaml (Multi-ML)

```
# let compo f g x = g (f x)
# type 'a list = Null | Node of 'a * 'a list
# let rec length l = match l with
    | Null → 0
    | Node (data,next) → 1+(length next);;
```

Q: What are the architectures BSP targets?

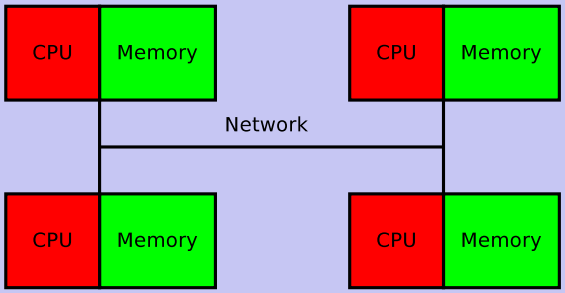
Distributed computing



Clusters

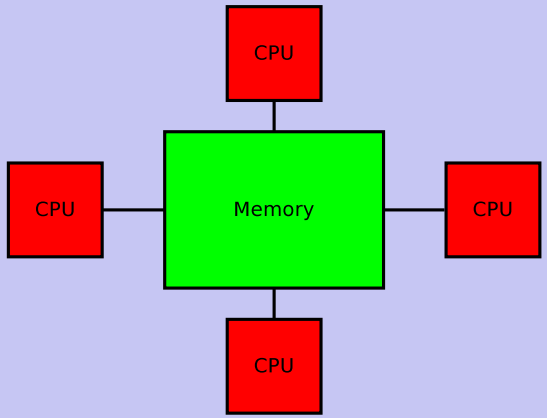
Q: What are the architectures BSP targets?

Distributed computing



Clusters

Shared memory



Multi-core

Q: What is Bulk Synchronous Parallelism? (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)

Properties:

Q: What is Bulk Synchronous Parallelism? (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)

Properties:

- "Confluent"
- Deterministic
- Deadlock-free

Q: What is Bulk Synchronous Parallelism? (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)

Properties:

- "Confluent"
- "Deadlock-free"
- Cost model (BSP)

Q: What is Bulk Synchronous Parallelism? (BSP)

The BSP computer

Defined by:

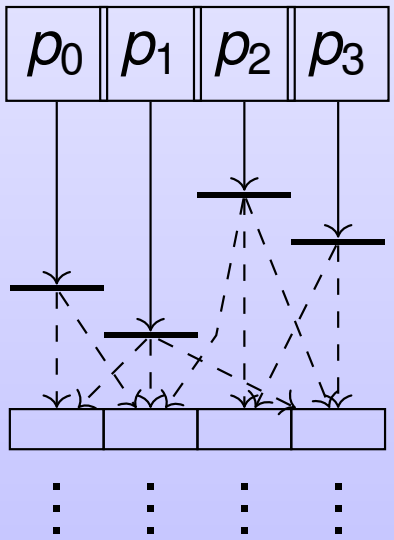
- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)

Algorithms

Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Cost model (p, g, L)



local computations

communication ($\otimes$**g**)

barrier ($\oplus$**L**)

next super-step

Q: What is Bulk Synchronous Parallelism? (BSP)

The BSP computer

Defined by:

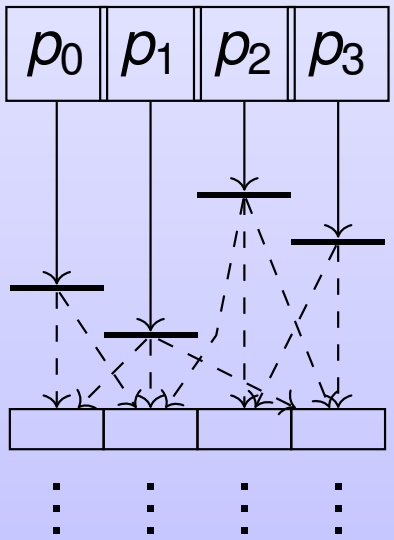
- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)

Algorithms

Super-steps execution

Properties:

- “Confluent”
- “Deadlock-free”
- Cost model (**p**, **g**, **L**)



local computations

communication ($\otimes$**g**)

barrier ($\oplus$**L**)

next super-step

Q: What is Bulk Synchronous Parallelism? (BSP)

The BSP computer

Defined by:

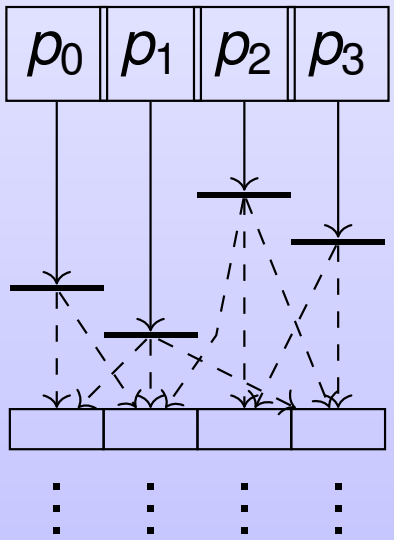
- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)

Algorithms

Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Cost model (**p**, **g**, **L**)



local computations

communication ($\otimes$**g**)

barrier ($\oplus$**L**)

next super-step

Q: What is the BSML language?

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaML

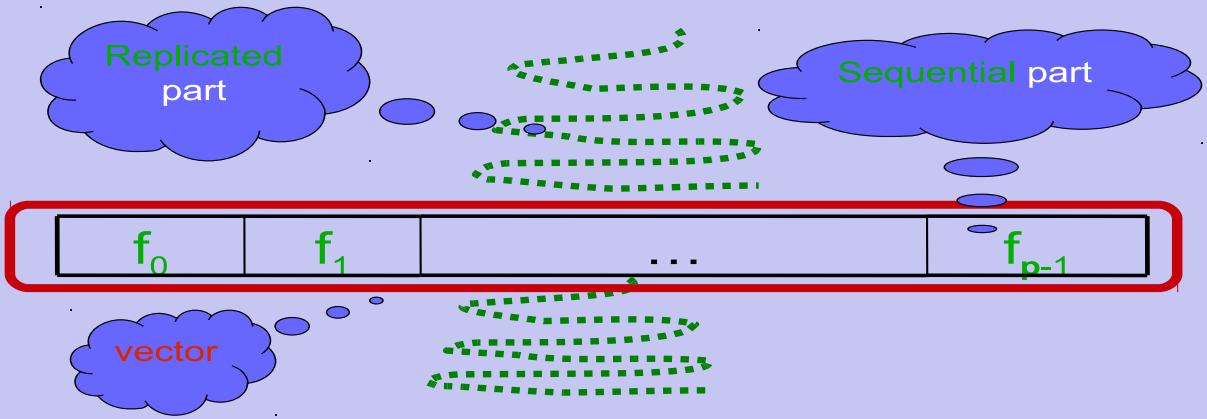
Q: What is the BSML language?

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaML

Main idea

Parallel data structure $\Rightarrow$ vectors:

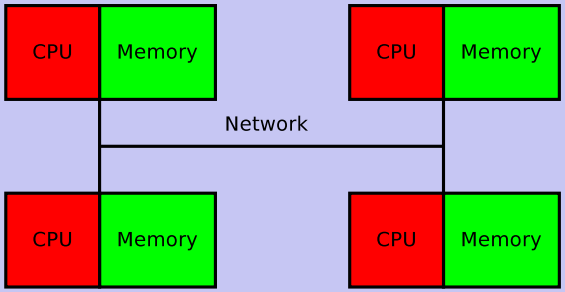


1. $\langle v_0, \ldots, v_{p-1} \rangle : \alpha\ par \equiv v_i$ on node i
2. Four primitives $\Rightarrow$ simple semantics

Outline

Q: What are the architectures Multi-BSP targets?

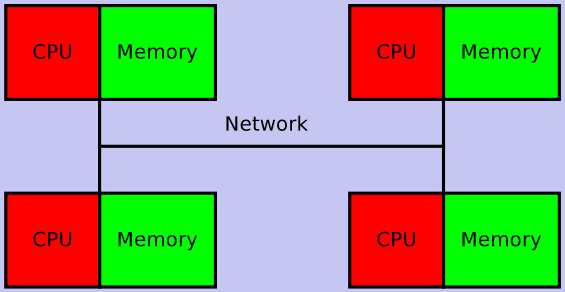
Distributed computing



Clusters

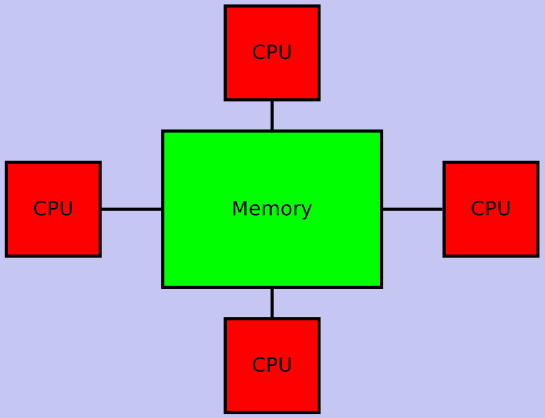
Q: What are the architectures Multi-BSP targets?

Distributed computing



Clusters

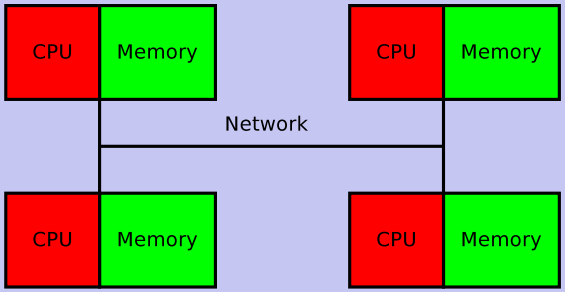
Shared memory



Multi-core

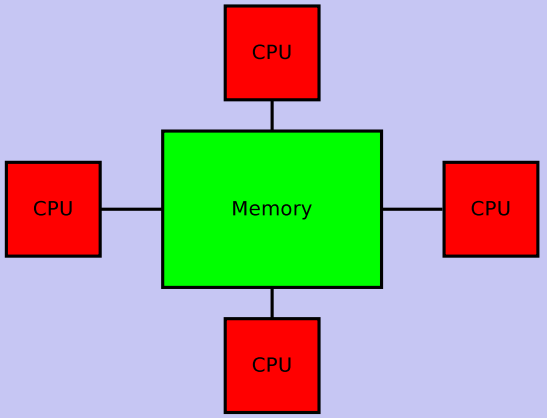
Q: What are the architectures Multi-BSP targets?

Distributed computing



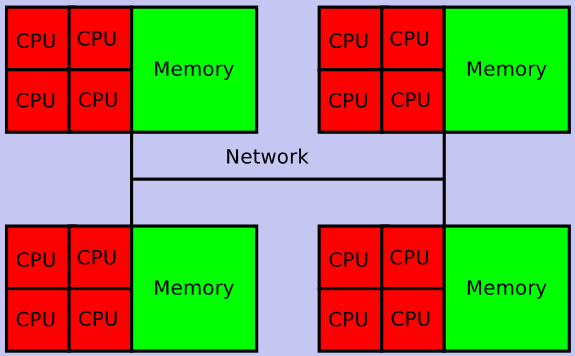
Clusters

Shared memory



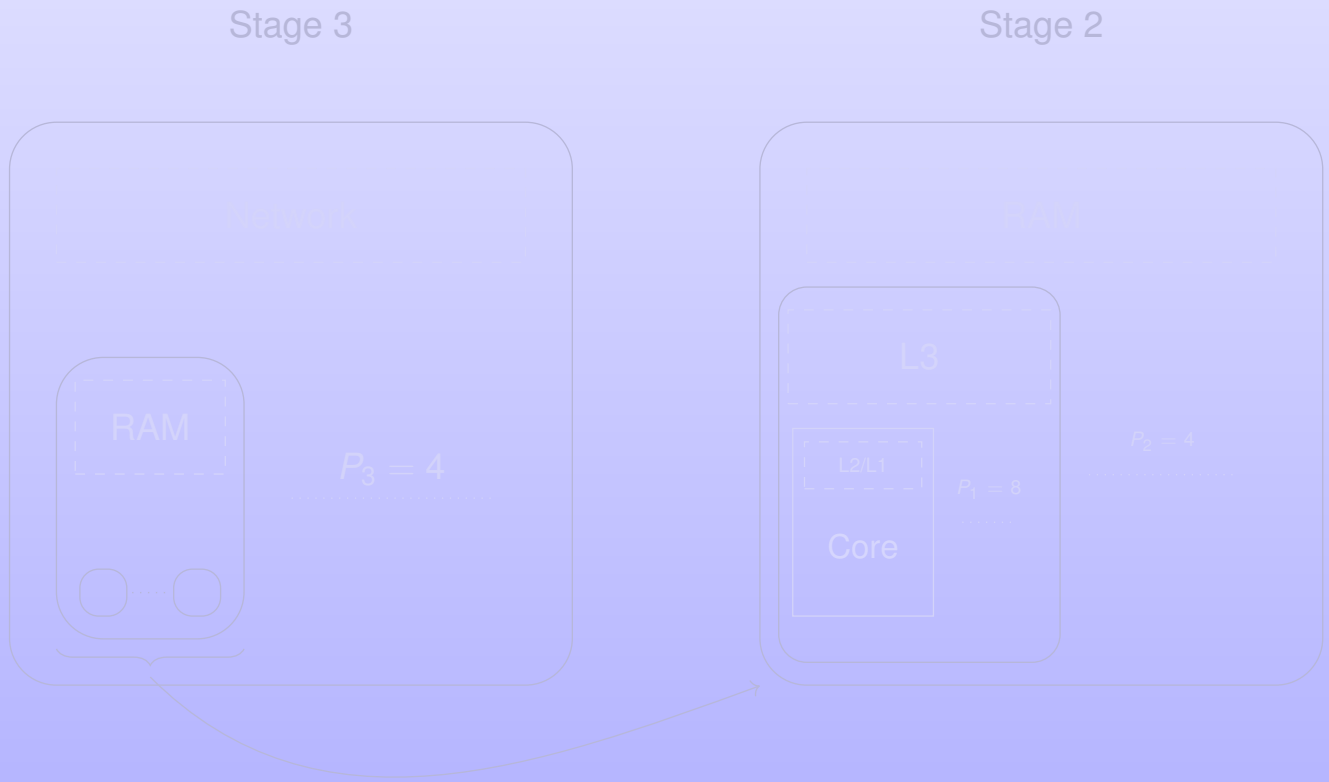
Multi-core

Hybrid model



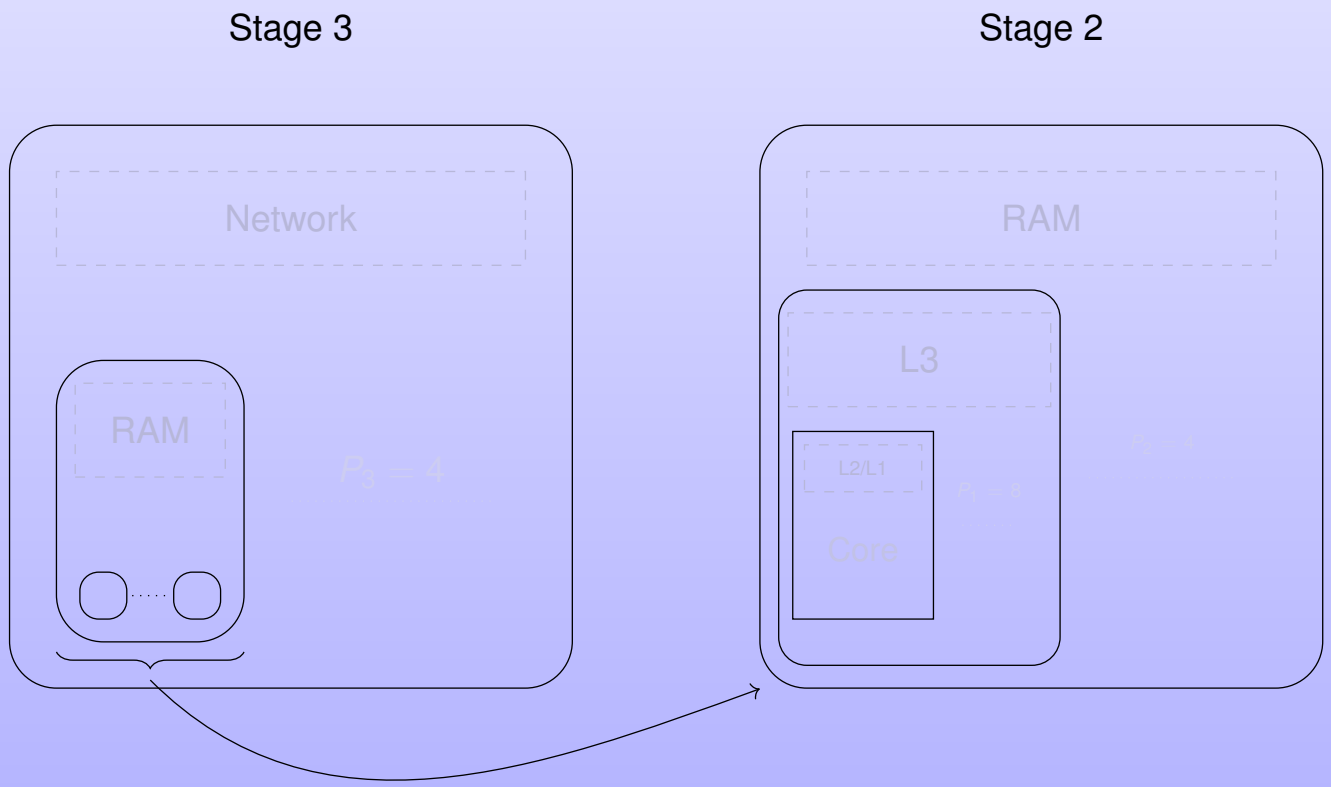
Q: What is MULTI-BSP?

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



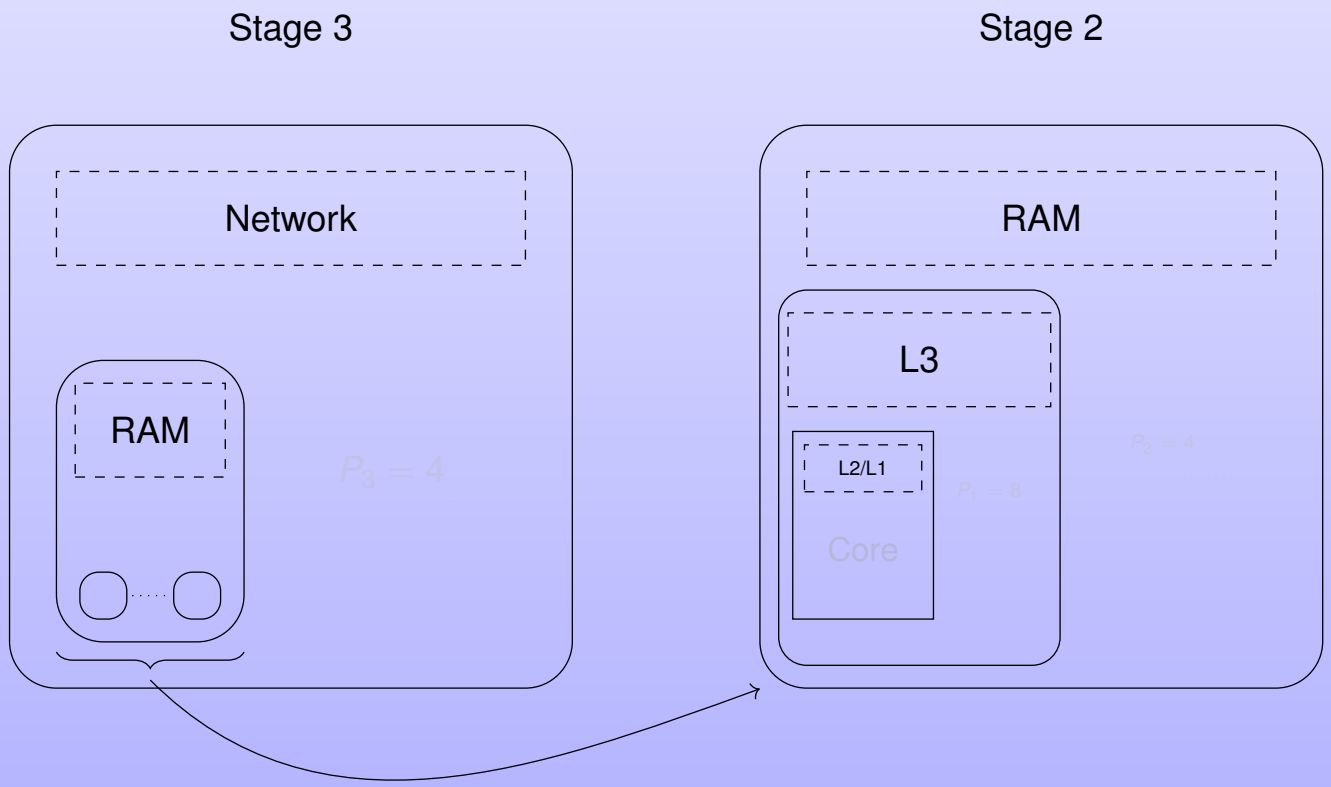
Q: What is MULTI-BSP?

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



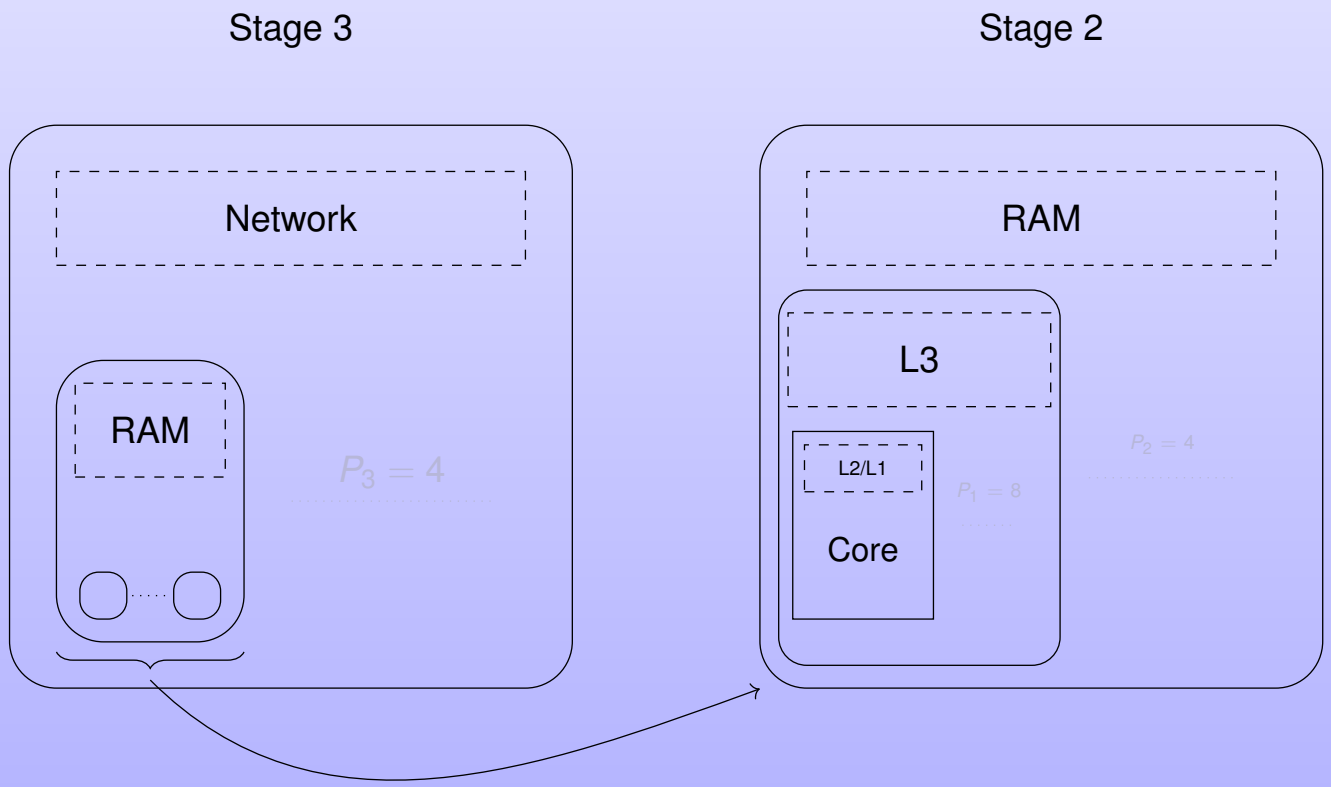
Q: What is MULTI-BSP?

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



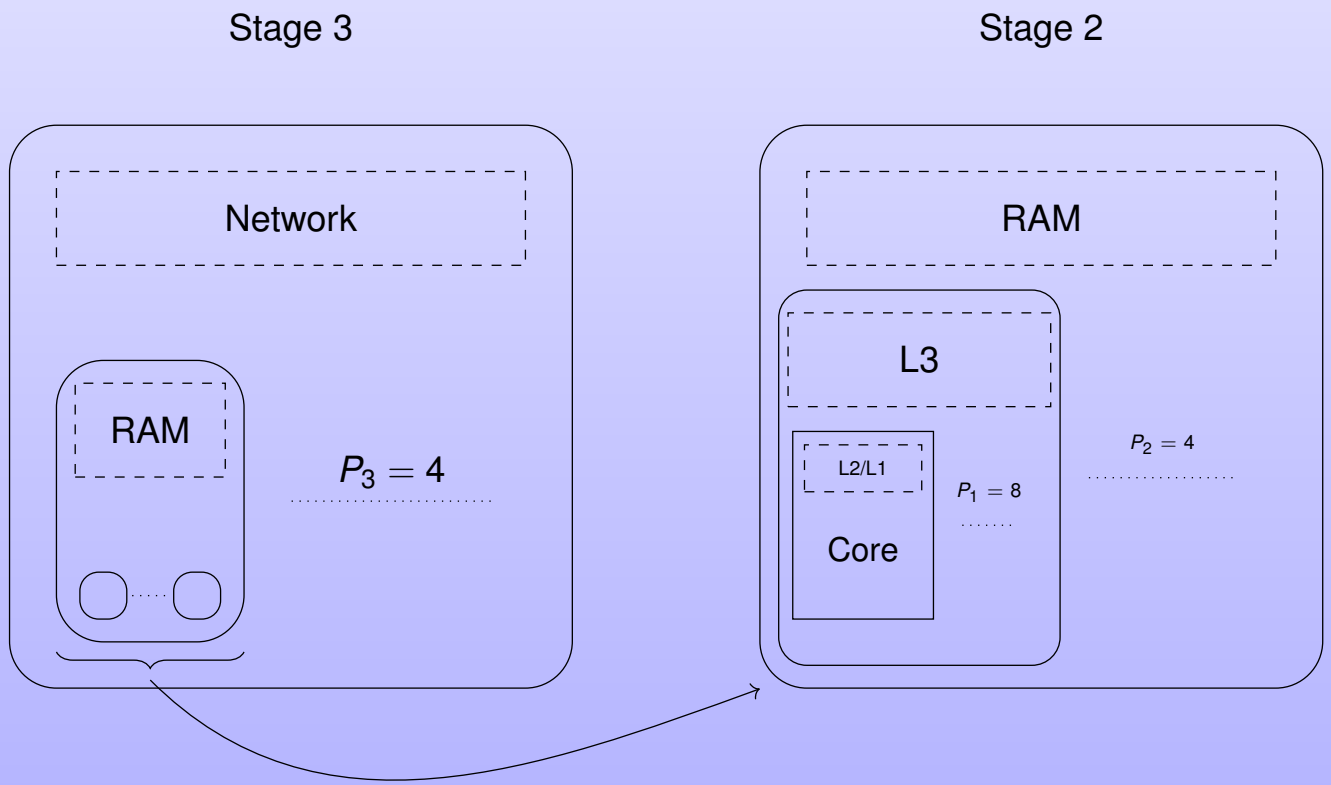
Q: What is MULTI-BSP?

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



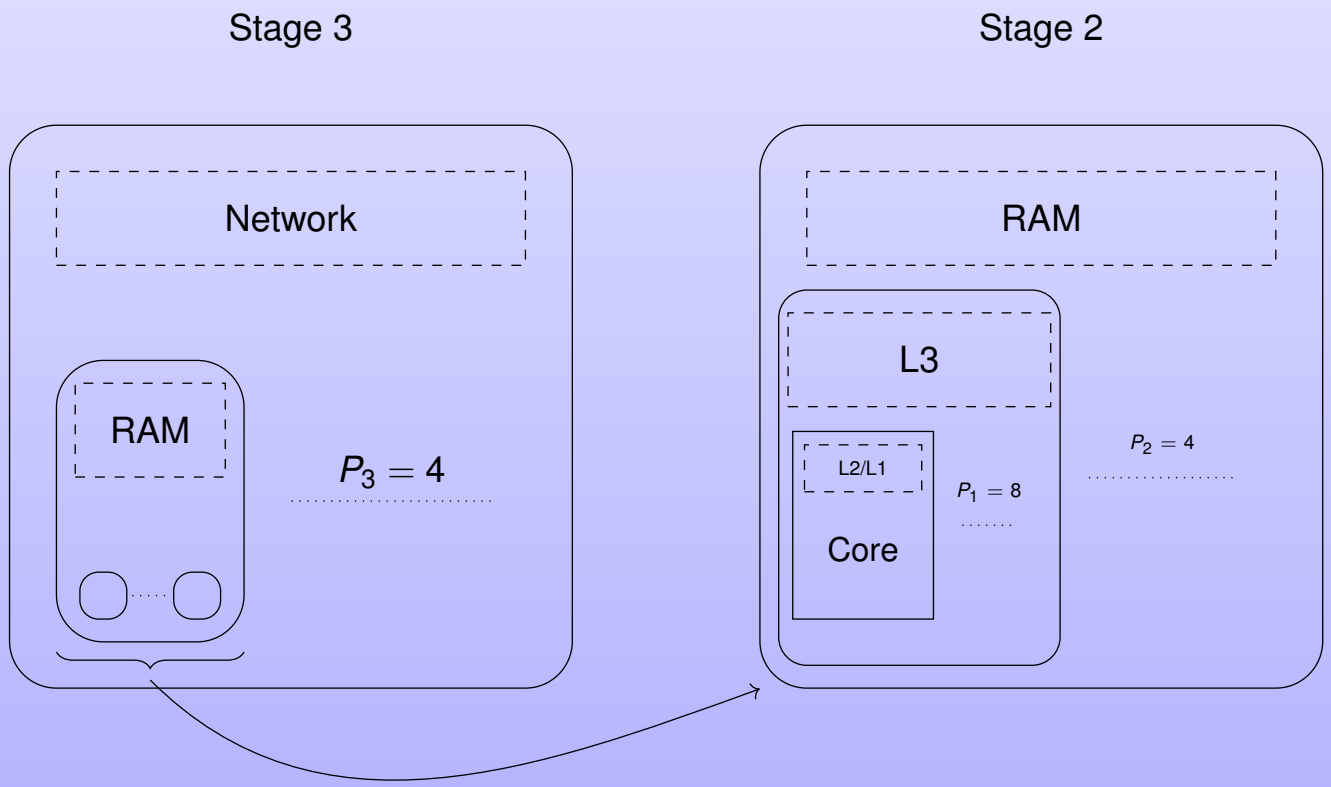
Q: What is MULTI-BSP?

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

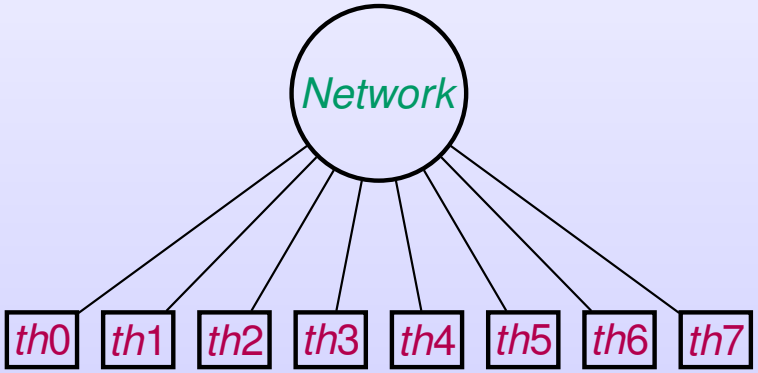


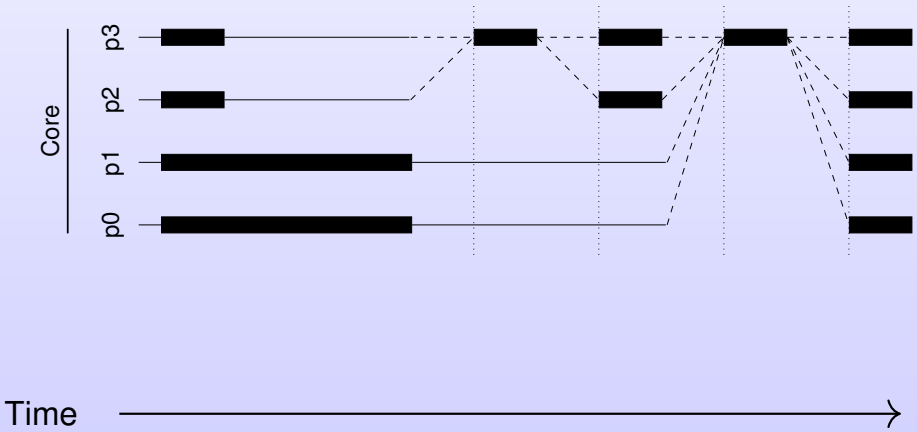
Q: What is MULTI-BSP?

- 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

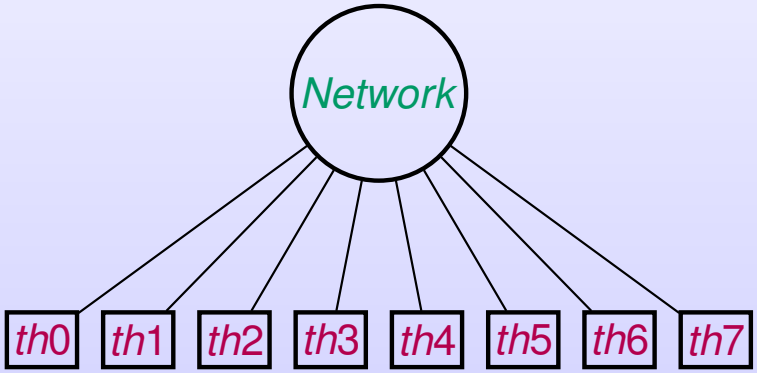


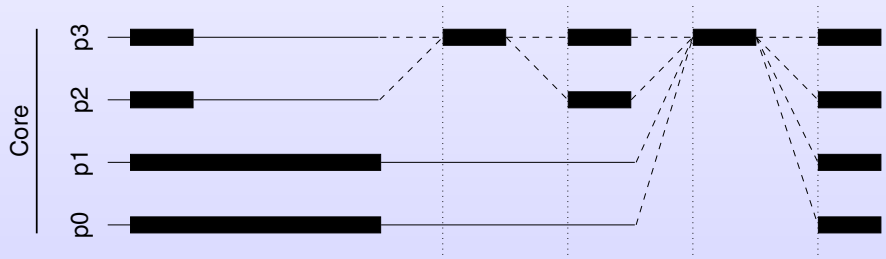
Q: BSP vs. MULTI-BSP?



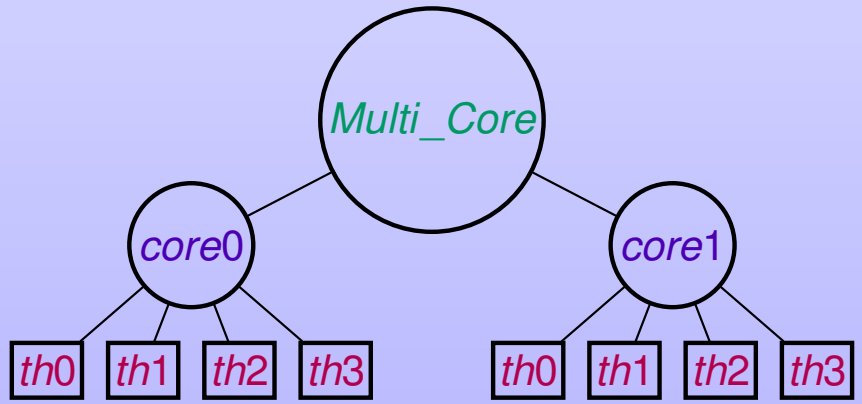


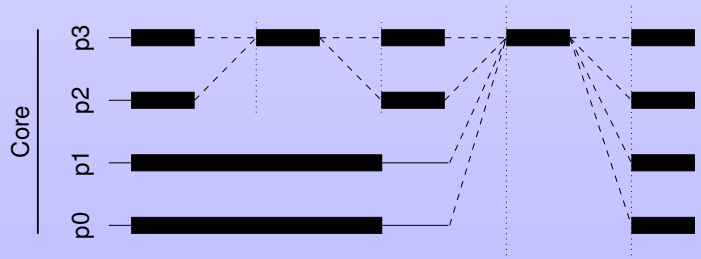
Q: BSP vs. MULTI-BSP?





Time





Q: What is the MULTI-ML language?

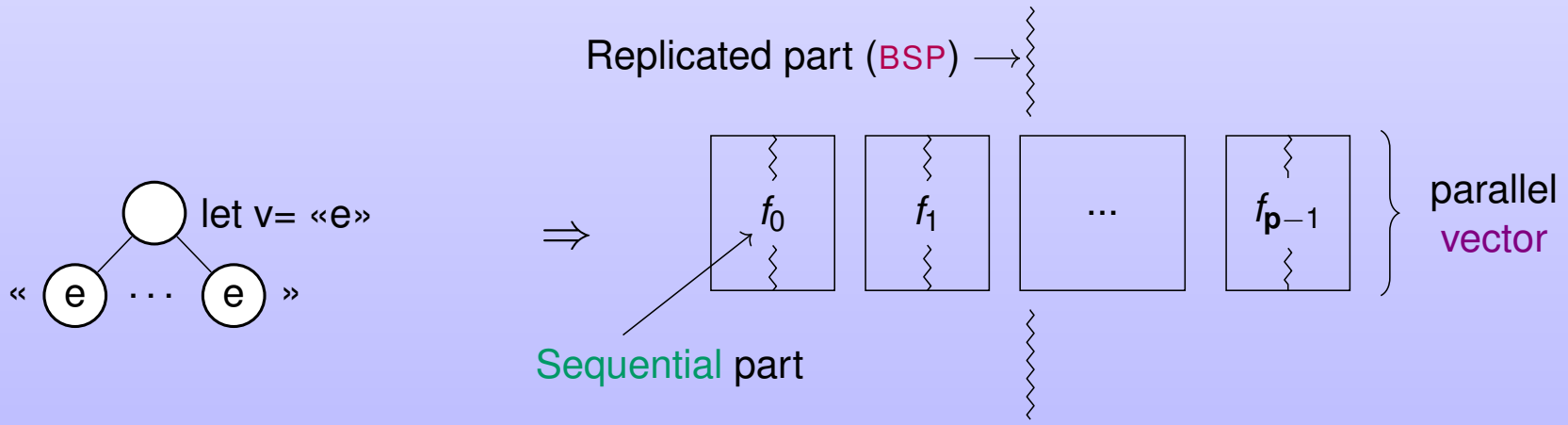
Basic ideas

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

Q: What is the MULTI-ML language?

Basic ideas

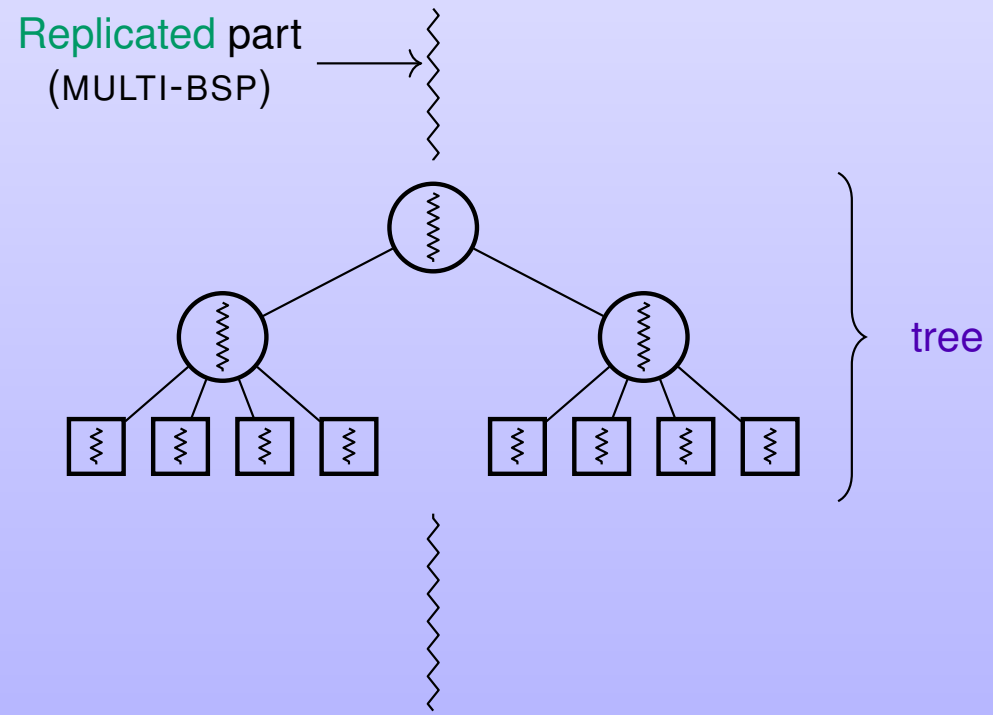
- BSML code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming



Q: What is the MULTI-ML language?

Basic ideas

- BSML 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



Q: What is tree recursion?

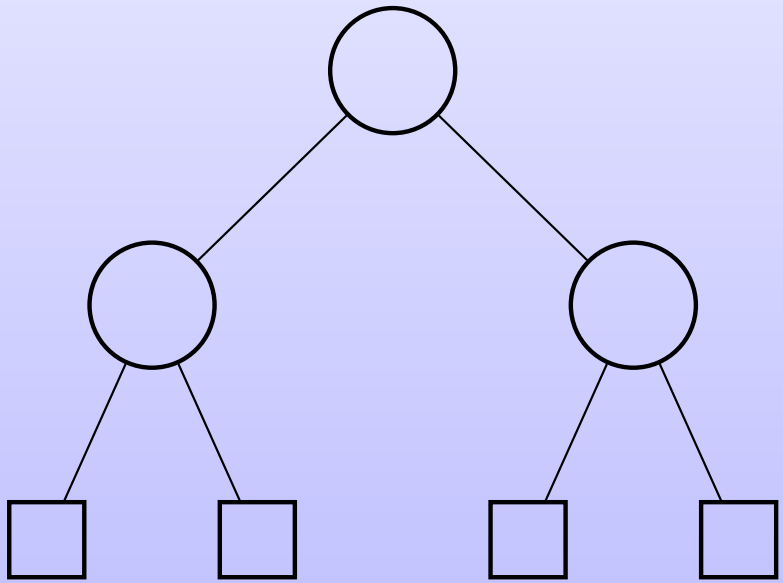
Recursion structure

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

Q: What is tree recursion?

Recursion structure

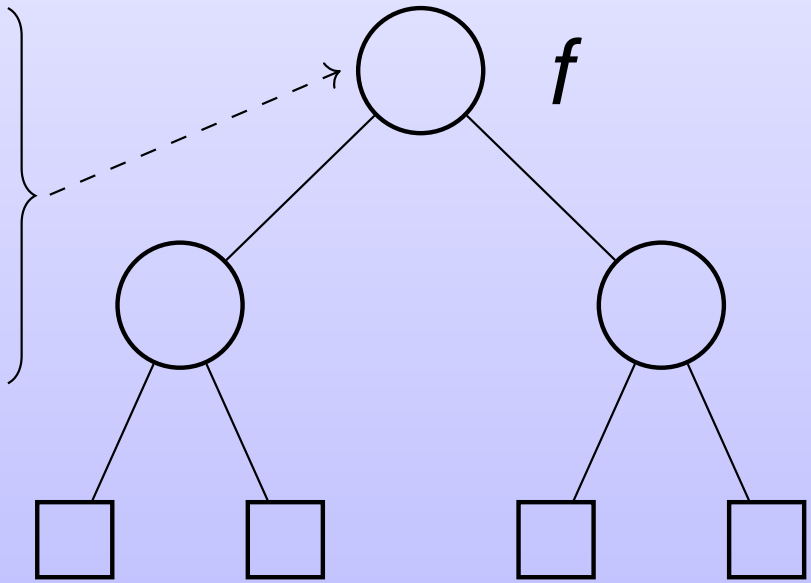
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



Q: What is tree recursion?

Recursion structure

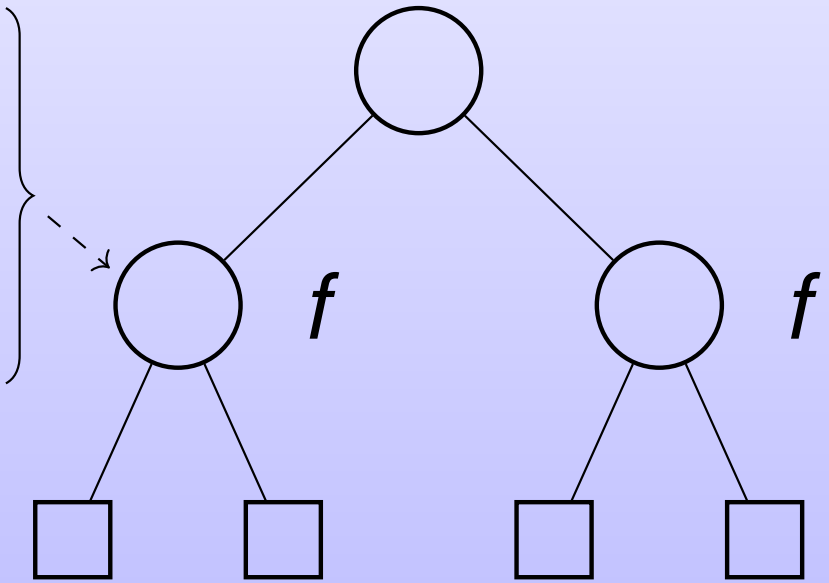
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



Q: What is tree recursion?

Recursion structure

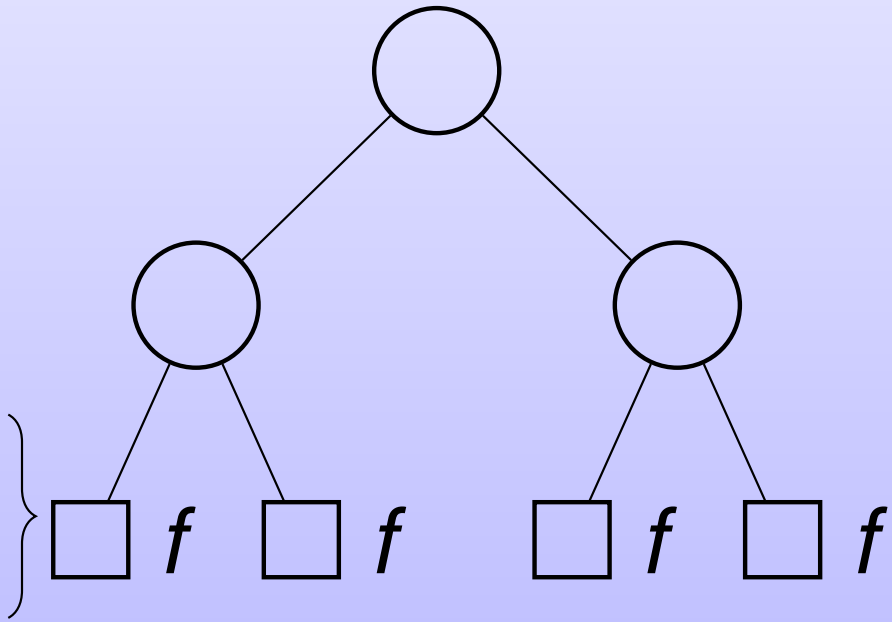
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



Q: What is tree recursion?

Recursion structure

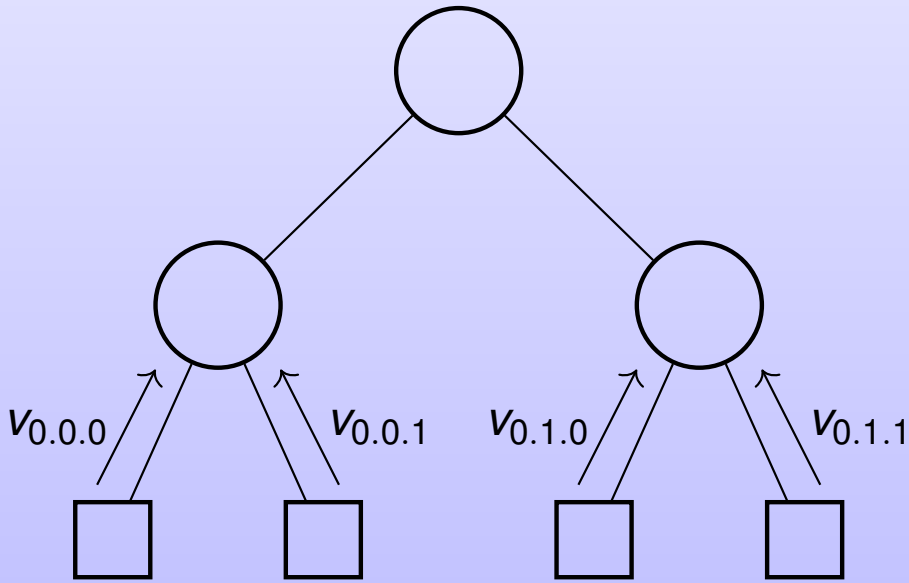
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



Q: What is tree recursion?

Recursion structure

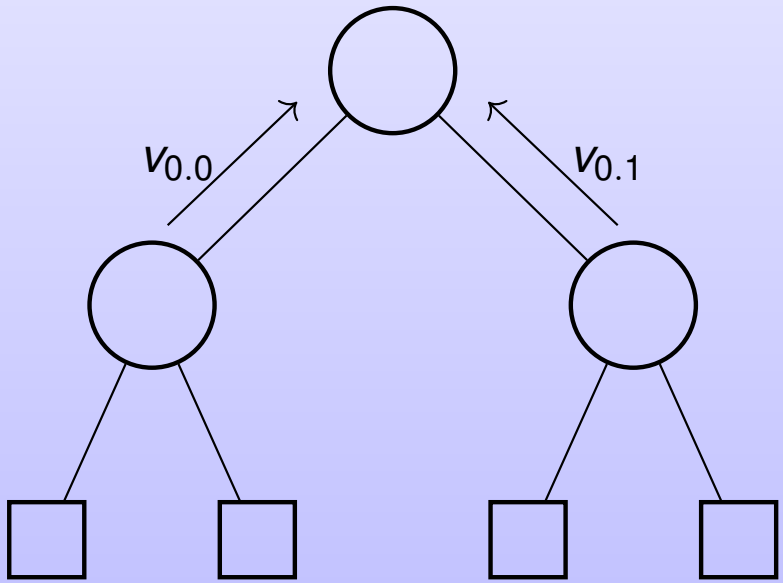
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



Q: What is tree recursion?

Recursion structure

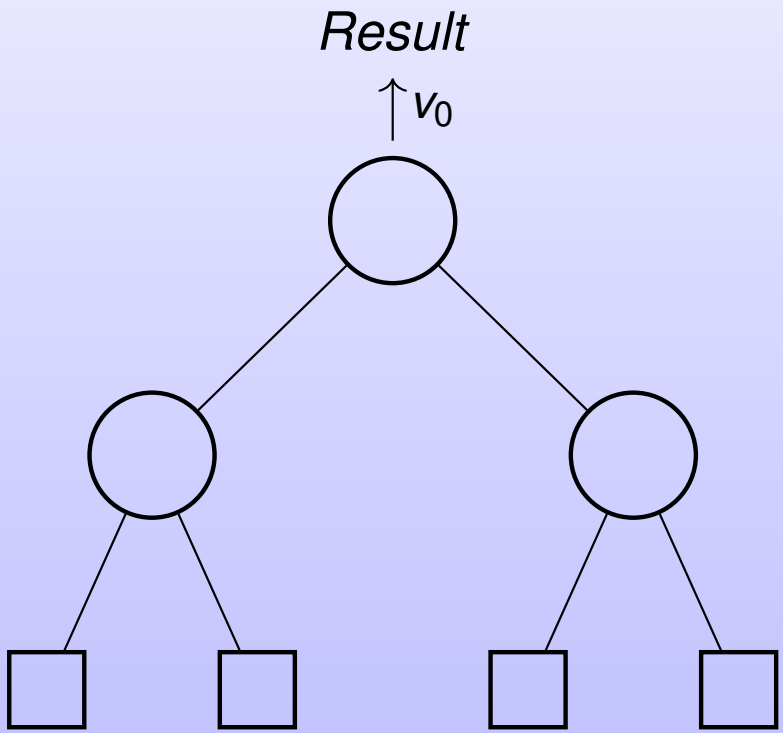
```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```



Q: What is tree recursion?

Recursion structure

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



Outline

Q: What are the errors?

Parallel program safety

- Replicated coherency

Replicated coherency

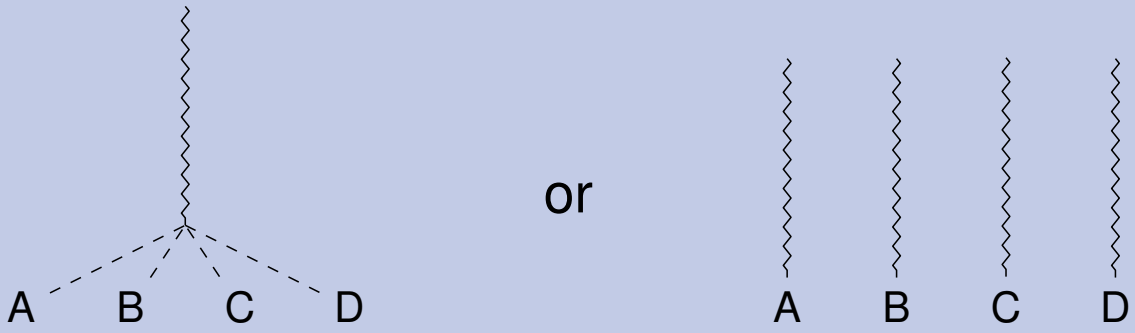
```
if random_bool () then
  multi_fct ()
else
  (fun _ → ...) ()
```

Q: What are the errors?

Parallel program safety

- Replicated coherency

Why ?



Replica

```
if random_bool () then
  multi_fct ()
else
  (fun _ → ...) ()
```

Q: What are the errors?

Parallel program safety

- Replicated coherency
- Level (memory) compatibility

Level (memory) compatibility

let v = ≪ **let multi** f x = ... ≫

let z = v+$**pid**$ // outside vectors

≪ v+$**pid**$ ≫ // outside nodes (e.g. inside leafs)

Q: What are the errors?

Parallel program safety

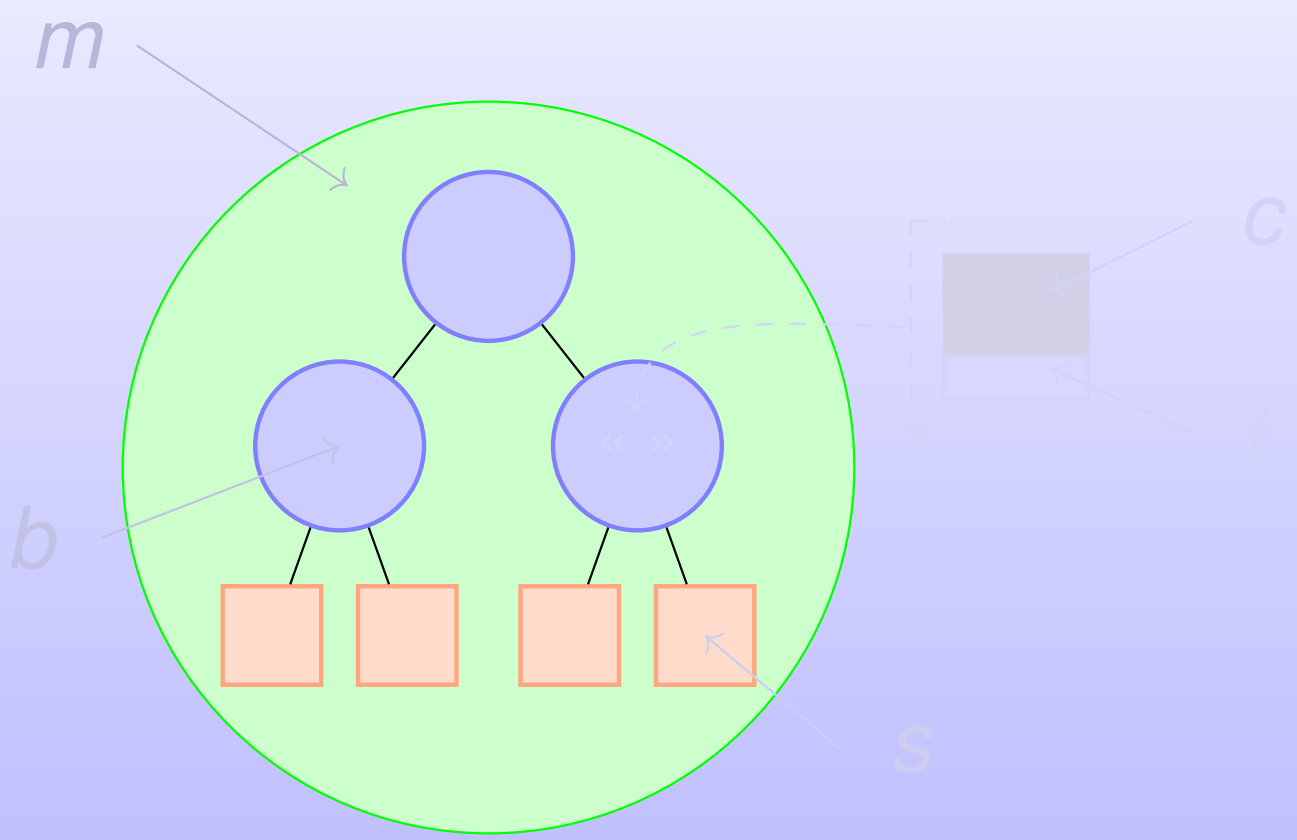
- Replicated coherency
- Level (memory) compatibility
- Control parallel structure imbrication
  - vector
  - tree

Parallel structure imbrication

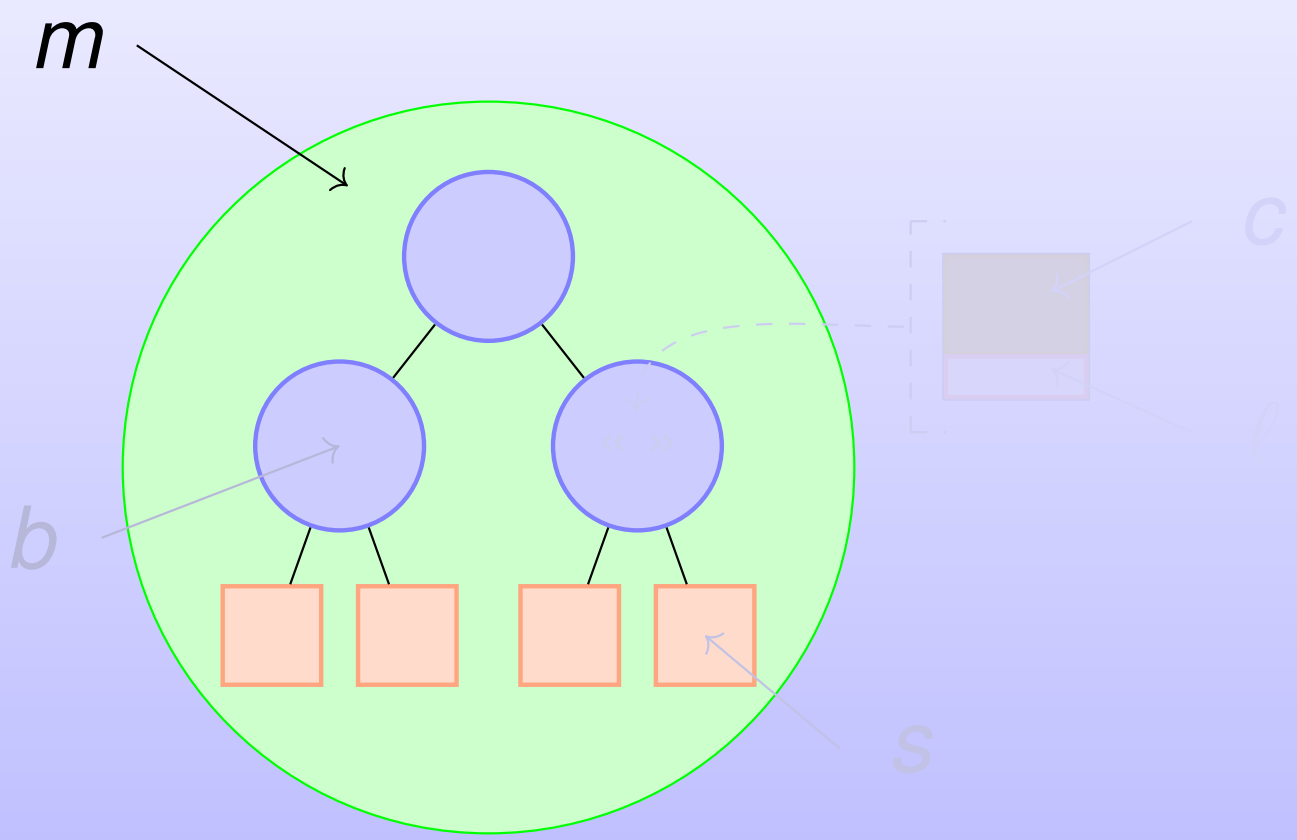
$\ll$ **let** v = $\ll$ 1 $\gg$ **in** v $\gg$

let v = $\ll$ 1 $\gg$ **in** $\ll$ v $\gg$

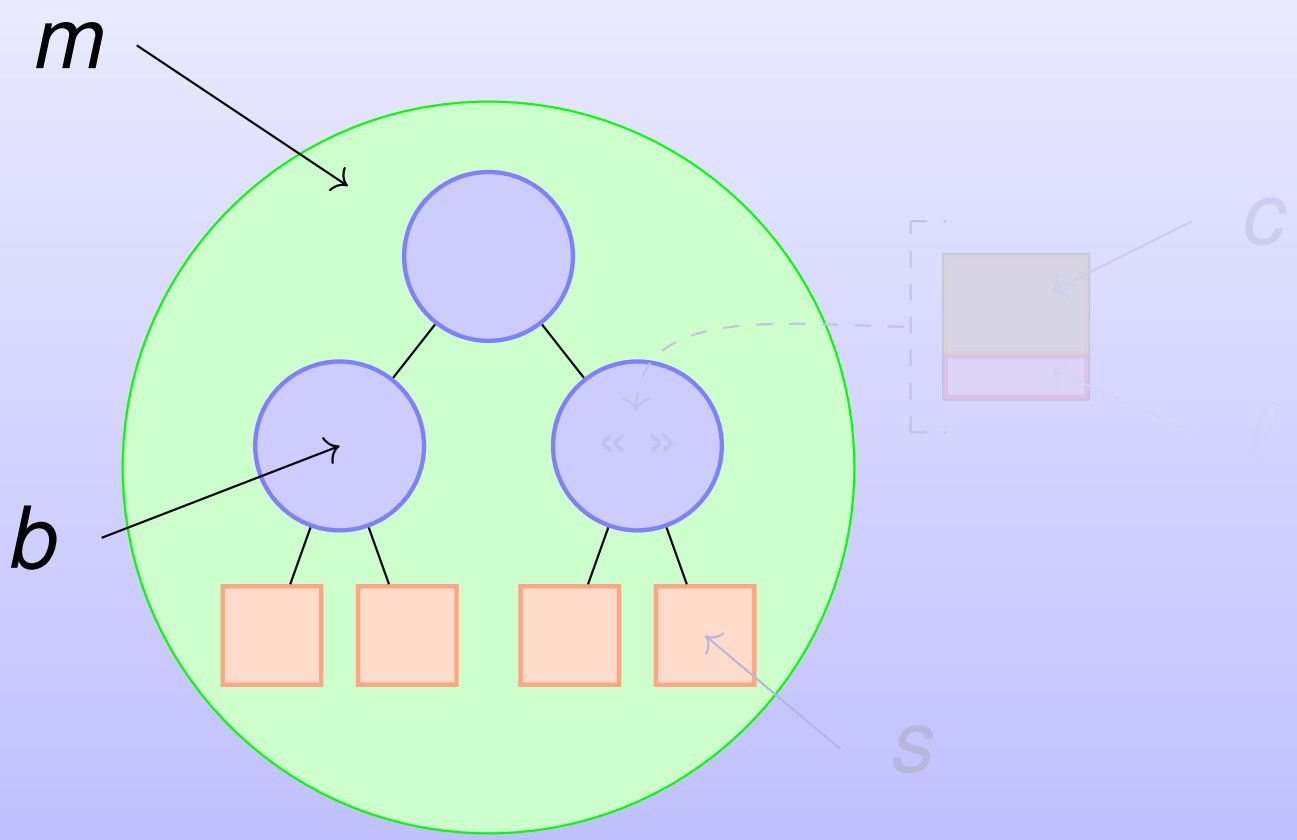
Type localities



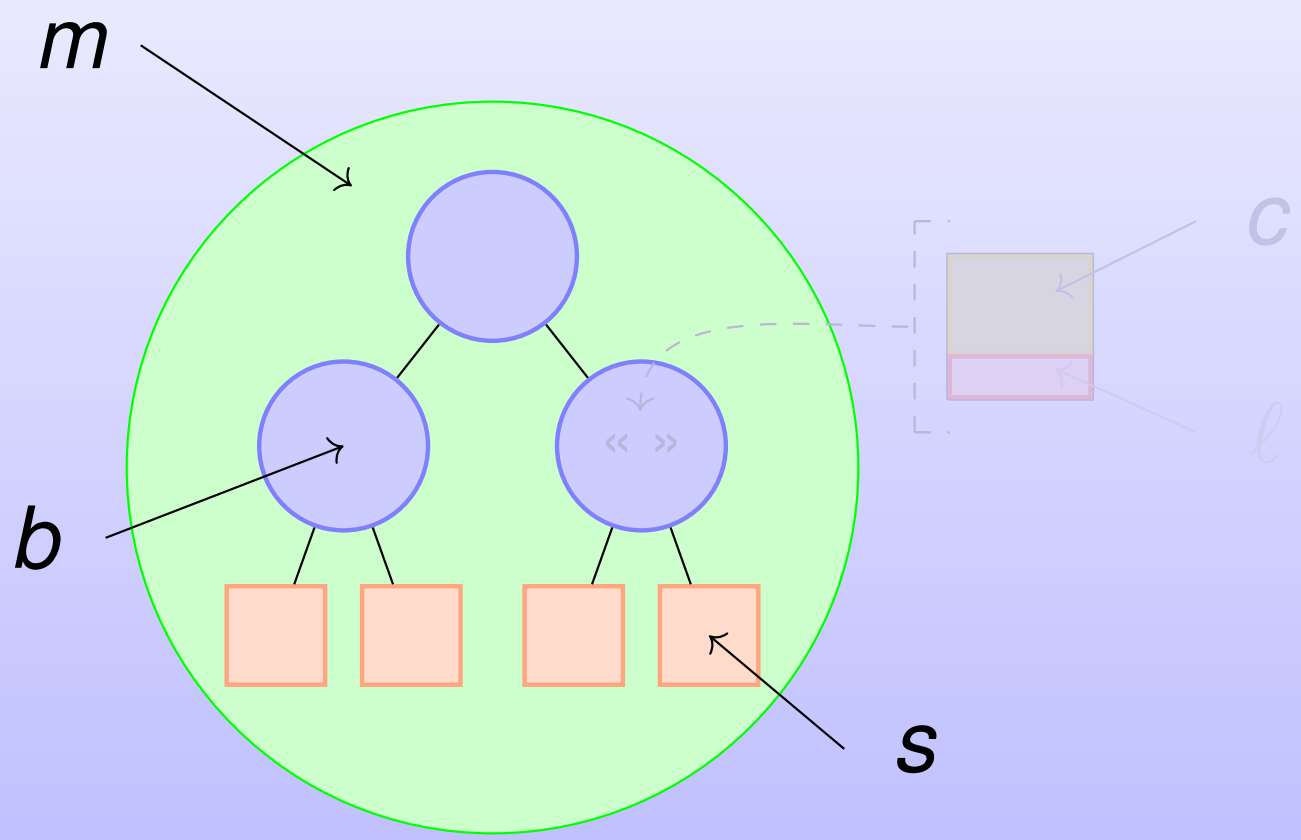
Type localities



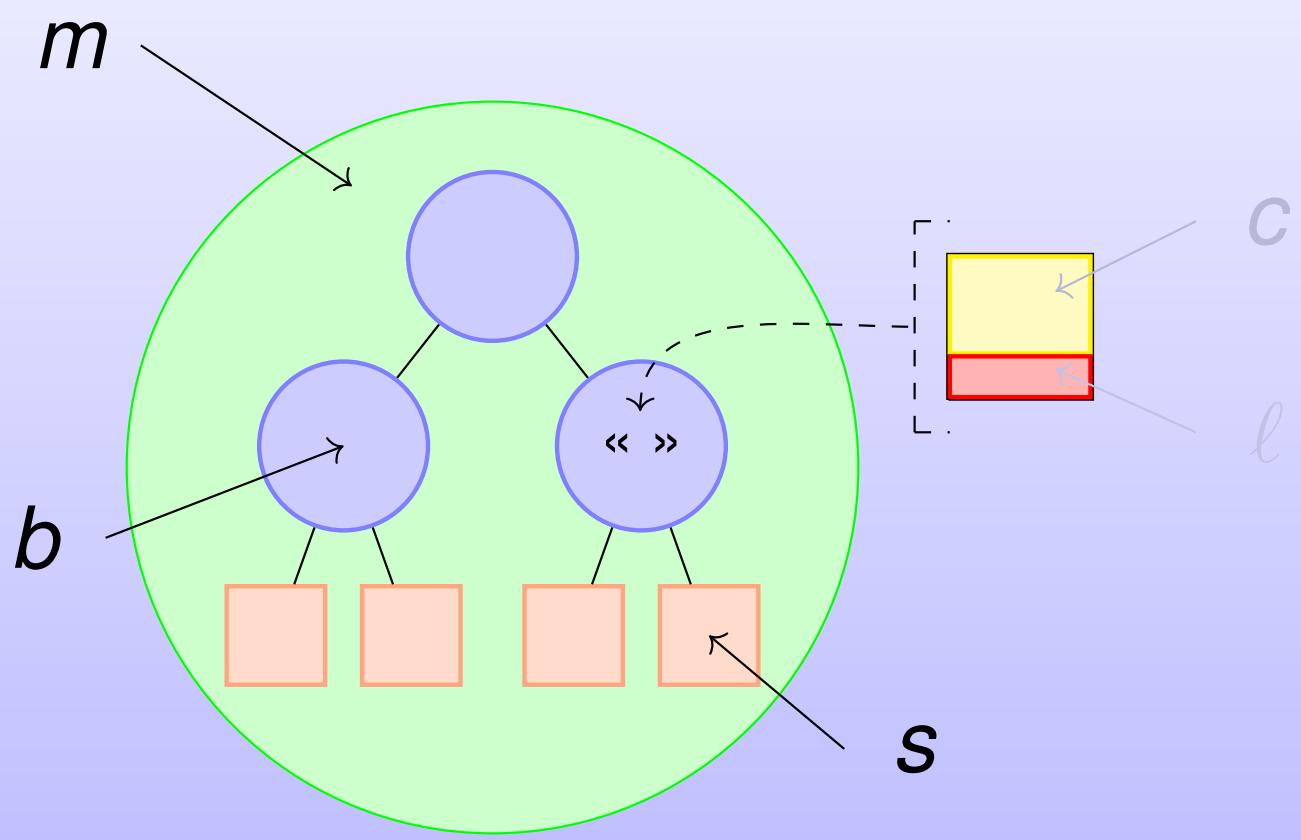
Type localities



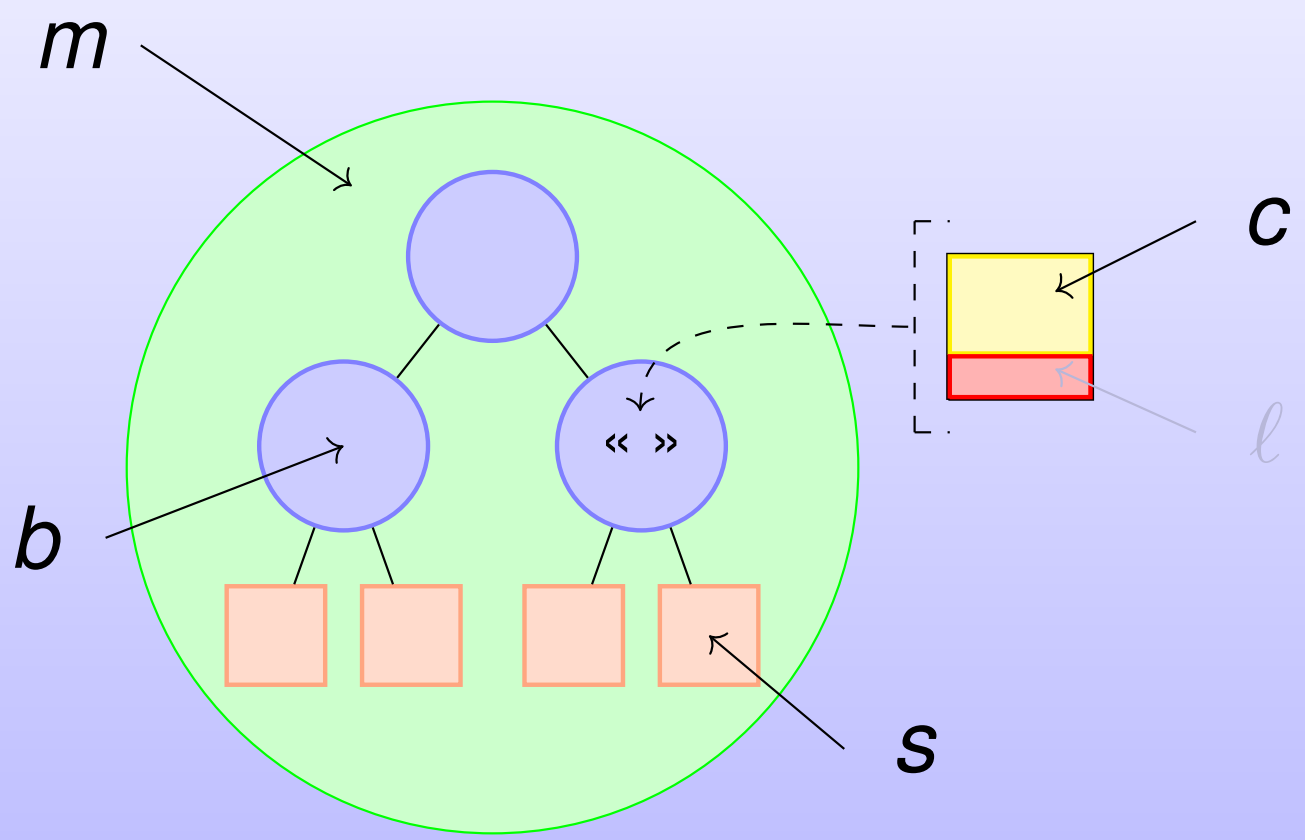
Type localities



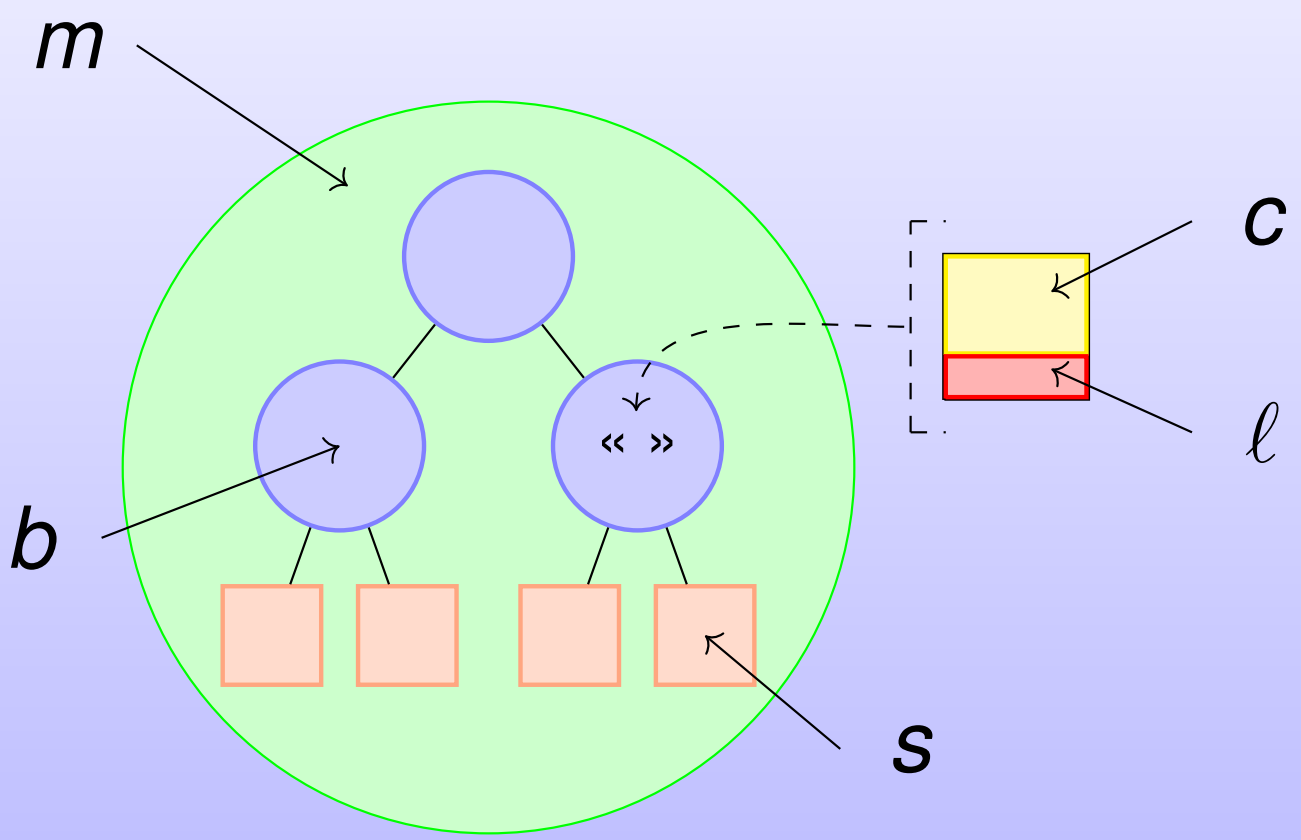
Type localities



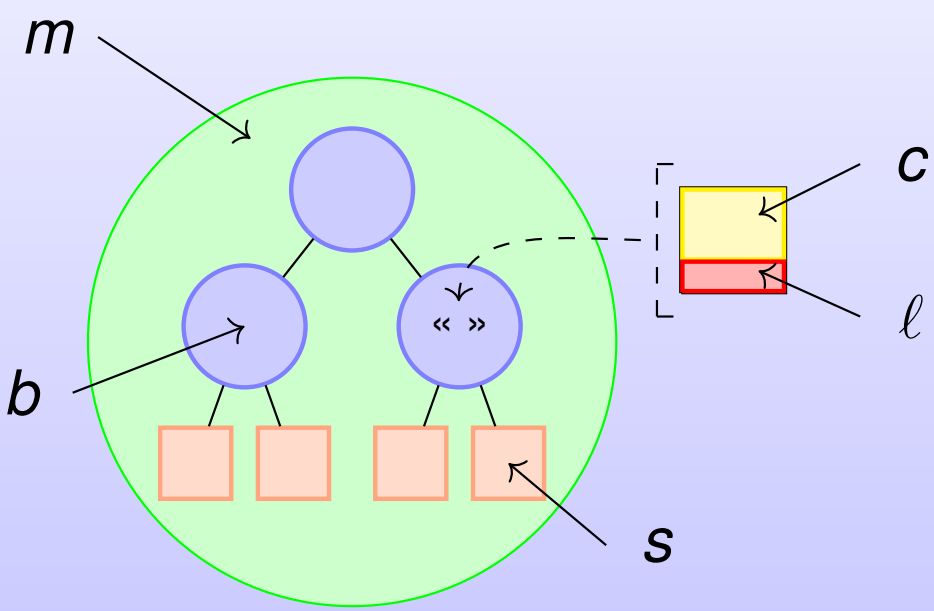
Type localities



Type localities



Type annotations



Type grammar

τ	$::=$	
	α_π	*type variable*
	$\texttt{Base}_\pi$	*base type*
	$(\tau, \tau)_\pi$	*pair*
	$\tau\ \texttt{Par}_b$	*vector*
	$\tau\ \texttt{Tree}_\pi$	*tree*
	$(\tau \xrightarrow{\pi} \tau)_\pi$	*arrow type*

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

Q: What are the types?

Latent effect as annotations

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

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

A BSP function

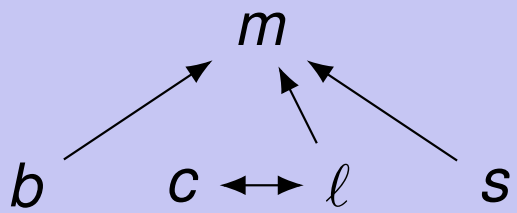
```
# let f = fun x →
    let v = ≪ ... ≫ in 1

−: val f : ('a_'z −(b)→ int_b)_m
```

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

Q: What are the constraints?

Definability: $\blacktriangleleft$

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

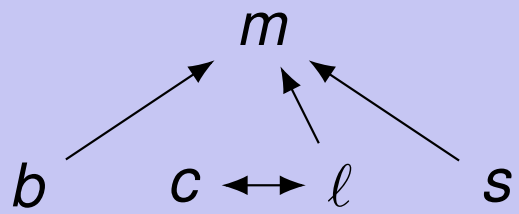


$\lambda_1 \blacktriangleleft \lambda_2$: "λ_1 *can be defined in* λ_2 *memory*"

Q: What are the constraints?

Definability: ◀

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



$\lambda_1 \blacktriangleleft \lambda_2$: "λ_1 *can be defined in* λ_2 *memory*"

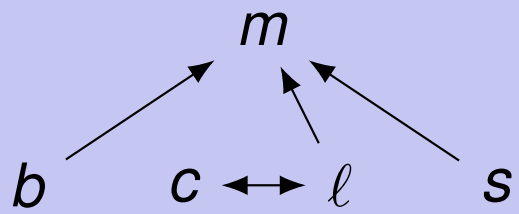
Example:

« let multi f x = ... » $\rightsquigarrow m \blacktriangleleft c$

Q: What are the constraints?

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:

« let multi f x = ... » $\rightsquigarrow m \blacktriangleleft c$

Error

Q: What are the rules?

Other relations

- Accessibility: "λ_1 can *read* in λ_2 memory"
- Propagation: prevailing effect among types
- Weakening: type generalization under constraints
- Serialisation: safety of communicating τ_π to locality λ

One example of rule (**let** $x = e_1$ **in** e_2)

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

Formal properties

Operational semantics

- Big step semantics $\Downarrow_p^{\mathcal{L}}$
- Big step semantics for diverging terms
- Programs that *"do not go wrong"*, $e \Rightarrow_{safe}$:

Formal properties

Operational semantics

- Big step semantics $\Downarrow_p^{\mathcal{L}}$
- Big step semantics for diverging terms
- Programs that *"do not go wrong"*, $e \Rightarrow_{safe}$:

Formal properties

Operational semantics

- Big step semantics $\Downarrow_p^{\mathcal{L}}$
- Big step semantics for diverging terms
- Programs that *"do not go wrong"*, $e \Rightarrow_{safe}$:
  - $\exists v.\ e \Downarrow_p^{\mathcal{L}} v$
  - $e \Uparrow_p^{\mathcal{L}}$

Formal properties

Operational semantics

- Big step semantics $\Downarrow_p^{\mathcal{L}}$
- Big step semantics for diverging terms
- Programs that *"do not go wrong"*, $e \Rightarrow_{safe}$:
  - $\exists v.\ e \Downarrow_p^{\mathcal{L}} v$
  - $e \Downarrow_p^{\mathcal{L}} \infty$

Formal properties

Operational semantics

- Big step semantics $\Downarrow_p^{\mathcal{L}}$
- Big step semantics for diverging terms
- Programs that *"do not go wrong"*, $e \Rightarrow_{safe}$:
  - $\exists v.\ e \Downarrow_p^{\mathcal{L}} v$
  - $e \Downarrow_p^{\mathcal{L}} \infty$

Theorem: type safety of MULTI-ML programs

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

Then: $\Gamma \vdash e : \tau_\pi / \varepsilon[c]$ implies that $e \Rightarrow_{safe}$

Formal properties

Operational semantics

- Big step semantics $\Downarrow_p^{\mathcal{L}}$
- Big step semantics for diverging terms
- Programs that *"do not go wrong"*, $e \Rightarrow_{safe}$:
  - $\exists v.\ \ e \Downarrow_p^{\mathcal{L}} v$
  - $e \Downarrow_p^{\mathcal{L}} \infty$

Theorem: type safety of MULTI-ML programs

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

Then: $\Gamma \vdash e : \tau_\pi / \varepsilon[c]$ implies that $e \Rightarrow_{safe}$

Outline

Conclusion

Main results

- BSML for BSP;
- MULTI-ML for MULTI-BSP
- A type system (with constraints and effects) to forbid erronous MULTI-ML codes
- Formal semantics of a mini-language
- Type inference algorithm

Conclusion

Main results

- BSML for BSP;
- MULTI-ML for MULTI-BSP
- A type system (with constraints and effects) to forbid erronous MULTI-ML codes
- Formal semantics of a mini-language
- Type inference algorithm

Perspectives (Ongoing/Future Work)

- Implementation for a full language
- Static and automatic analysis for cost prediction
- Application to bigger examples

Merci !