

Domain Specific Formal Languages

Klaim

Francesco Tiezzi

University of Camerino
francesco.tiezzi@unicam.it

A.A. 2016/2017



Outline

- 1 Motivations
- 2 KLAIM syntax
- 3 KLAIM semantics
- 4 An example
- 5 KLAIM implementation

Original goal

Programming *Global Computers*: distributed software architecture models

- Large number of heterogeneous and autonomous components
- Wide area distribution, manifest to application programmers
- Different authorities with different administrative and security policies
- Components with highly dynamic behavior that must adapt to unpredictable changes over time of the environment, due e.g. to
 - lack of services
 - node failures
 - network reconfiguration
 - ...
- Mobile components must support heterogeneity and interoperability, as they may detach from a node and re-attach later on a different one

Opened, loosely coupled, possibly collaborative settings

Programming Global Computers

Programming Languages would definitely benefit from explicit primitives for

- **Concurrency**
 - considering parallel and non-deterministic computations
- **Distribution**
 - computing over different (explicit) localities
- **Mobility**
 - moving code and processes over localities

and from formal semantics and associated tools for

- **Reasoning** on programs behaviour

The *spatial* dimension

- Many foundational languages have been designed that features mechanisms
 - for coordinating and monitoring the use of resources
 - for code and process mobility
 - for managing security
- All these foundational languages encompass a notion of *location*
 - to reflect the idea of *administrative domains*
 - to model the fact that computations at a certain location are under the control of a specific authority
- They focus on the *spatial* dimension of GC programming (which is often referred to as *network awareness*)

Card game



KLAIM design goal

Developing a simple *programming language* and associated tools for **network aware** and **mobile applications** with a tractable semantic theory that permits programs verification

Our Starting Points (1980 — ...)

- Process Algebras
 - CCS, CSP, ...
- Calculi and languages for Mobility
 - Pi-calculus, Obliq, Ambients, ...
- Tuple-based Interaction Models
 - Linda
- Modal and Temporal Logics
 - HML, CTL, ACTL, μ -calculus, ...

KLAIM: main ingredients

Kernel Language for Agent Interaction and Mobility
a formalism for concurrent, distributed and mobile systems

- *Linda* communication model
 - Asynchronous communication
 - Shared tuple spaces
 - Pattern Matching
- Explicit use of localities
 - Multiple distributed tuple spaces
 - Code and process mobility
- *Process Calculus* flavored
 - Small set of basic operators
 - Clean operational semantics

KLAIM: main ingredients

Kernel Language for Agent Interaction and Mobility
a formalism for concurrent, distributed and mobile systems

- *Linda* communication model
 - Asynchronous communication
 - Shared tuple spaces
 - Pattern Matching
- Explicit use of localities
 - Multiple distributed tuple spaces
 - Code and process mobility
- *Process Calculus* flavored
 - Small set of basic operators
 - Clean operational semantics

KLAIM: main ingredients

Kernel Language for Agent Interaction and Mobility
a formalism for concurrent, distributed and mobile systems

- *Linda* communication model
 - Asynchronous communication
 - Shared tuple spaces
 - Pattern Matching
- Explicit use of localities
 - Multiple distributed tuple spaces
 - Code and process mobility
- *Process Calculus* flavored
 - Small set of basic operators
 - Clean operational semantics

KLAIM: main ingredients

Kernel Language for Agent Interaction and Mobility
a formalism for concurrent, distributed and mobile systems

- *Linda* communication model
 - Asynchronous communication
 - Shared tuple spaces
 - Pattern Matching
- Explicit use of localities
 - Multiple distributed tuple spaces
 - Code and process mobility
- *Process Calculus* flavored
 - Small set of basic operators
 - Clean operational semantics

Tuples and Pattern Matching

- Tuples: $(\text{"foo"}, 10 + 5, \text{true})$
 - Actual Fields (i.e. expressions)
- Templates: $(!s, 15, !b)$
 - Actual Fields
 - Formal Fields (i.e. variables)
- *Pattern Matching*:
 - Formal fields match any field of the same type
 - Actual fields match if identical

Tuples and Pattern Matching

- Tuples: $(\text{"foo"}, 10 + 5, \text{true})$
 - Actual Fields (i.e. expressions)
- Templates: $(!s, 15, !b)$
 - Actual Fields
 - Formal Fields (i.e. variables)
- *Pattern Matching:*
 - Formal fields match any field of the same type
 - Actual fields match if identical

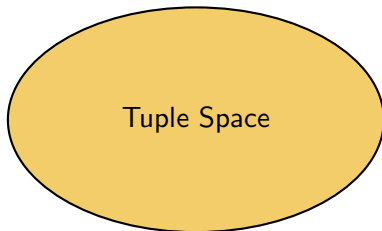
Tuples and Pattern Matching

- Tuples: $(\text{"foo"}, 10 + 5, \text{true})$
 - Actual Fields (i.e. expressions)
- Templates: $(!s, 15, !b)$
 - Actual Fields
 - Formal Fields (i.e. variables)
- *Pattern Matching*:
 - Formal fields match any field of the same type
 - Actual fields match if identical

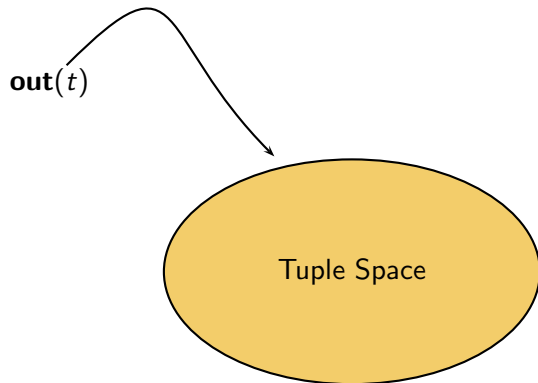
$(!s, 15, !b)$ **matches** $(\text{"foo"}, 15, \text{true})$

$(!s, 15, !b)$ does **not** match $(\text{"foo"}, 10, \text{true})$

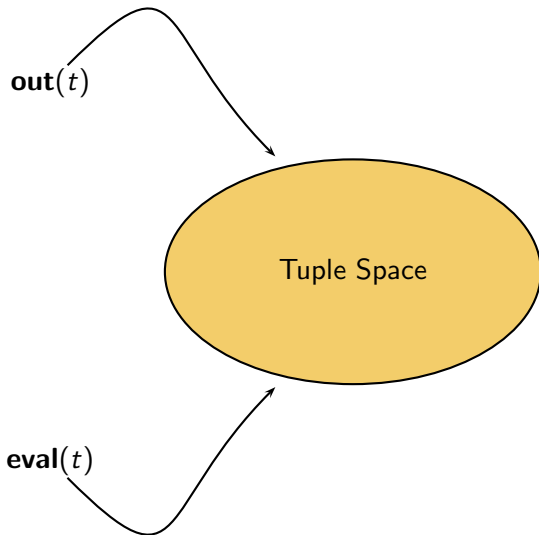
Linda communication model



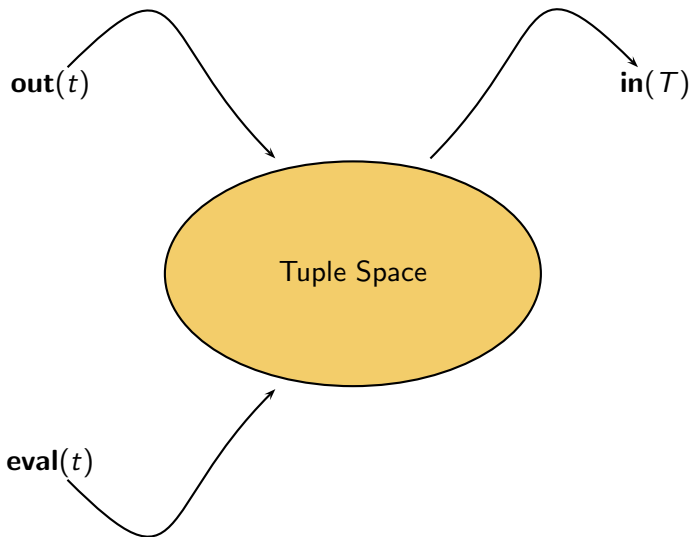
Linda communication model



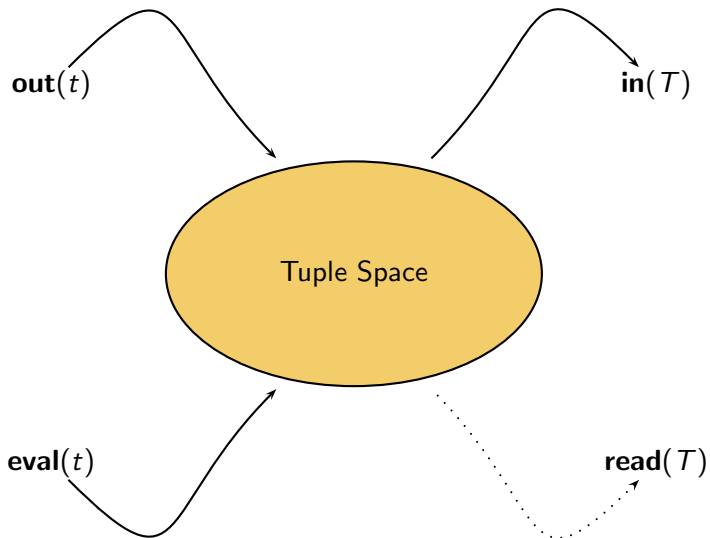
Linda communication model



Linda communication model



Linda communication model



From Linda to KLAIM

- *Localities* to model distribution
 - Physical localities (network addresses), i.e. names
 - Logical localities (aliases for addresses), i.e. variables
 - A distinct variable **self** indicates the node a process is on
- Local and remote *operations*
to read/withdraw/write tuples, spawn processes, create nodes
- *Allocation environments* to associate physical to logical localities
 - Logical localities are local aliases for network addresses
 - Programmers may not know the exact network topology
- *Process Calculi* operators to specify behaviours
 - Action prefix
 - Parallel composition
 - Name creation / scope delimitation

From Linda to KLAIM

- *Localities* to model distribution
 - Physical localities (network addresses), i.e. names
 - Logical localities (aliases for addresses), i.e. variables
 - A distinct variable **self** indicates the node a process is on
- Local and remote *operations*
to read/withdraw/write tuples, spawn processes, create nodes
- *Allocation environments* to associate physical to logical localities
 - Logical localities are local aliases for network addresses
 - Programmers may not know the exact network topology
- *Process Calculi* operators to specify behaviours
 - Action prefix
 - Parallel composition
 - Name creation / scope delimitation

From Linda to KLAIM

- *Localities* to model distribution
 - Physical localities (network addresses), i.e. names
 - Logical localities (aliases for addresses), i.e. variables
 - A distinct variable **self** indicates the node a process is on
- Local and remote *operations*
to read/withdraw/write tuples, spawn processes, create nodes
- *Allocation environments* to associate physical to logical localities
 - Logical localities are local aliases for network addresses
 - Programmers may not know the exact network topology
- *Process Calculi* operators to specify behaviours
 - Action prefix
 - Parallel composition
 - Name creation / scope delimitation

From Linda to KLAIM

- *Localities* to model distribution
 - Physical localities (network addresses), i.e. names
 - Logical localities (aliases for addresses), i.e. variables
 - A distinct variable **self** indicates the node a process is on
- Local and remote *operations*
to read/withdraw/write tuples, spawn processes, create nodes
- *Allocation environments* to associate physical to logical localities
 - Logical localities are local aliases for network addresses
 - Programmers may not know the exact network topology
- *Process Calculi* operators to specify behaviours
 - Action prefix
 - Parallel composition
 - Name creation / scope delimitation

KLAIM

- KLAIM consists of three layers: nets, processes, and actions
- *Nets* specify the overall structure of a system, including
 - where processes and tuple spaces are located
 - how nodes are interconnected
- *Processes* are the actors in this system and execute by performing *actions*

KLAIM

- KLAIM consists of three layers: nets, processes, and actions
- *Nets* specify the overall structure of a system, including
 - where processes and tuple spaces are located
 - how nodes are interconnected
- *Processes* are the actors in this system and execute by performing *actions*

KLAIM

- KLAIM consists of three layers: nets, processes, and actions
- *Nets* specify the overall structure of a system, including
 - where processes and tuple spaces are located
 - how nodes are interconnected
- *Processes* are the actors in this system and execute by performing *actions*

KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



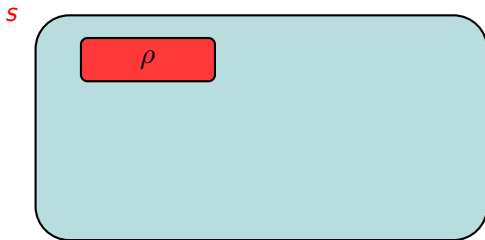
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



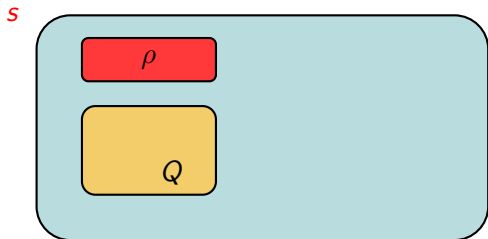
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



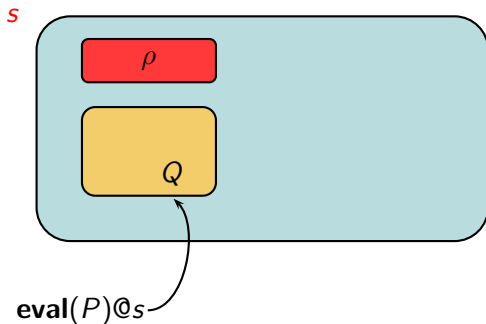
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



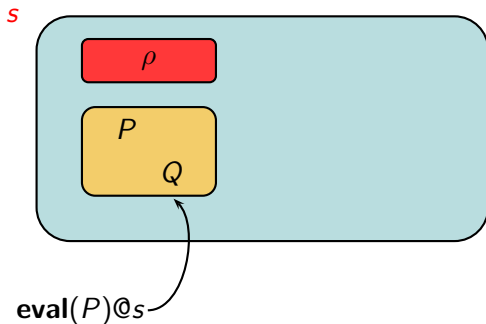
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



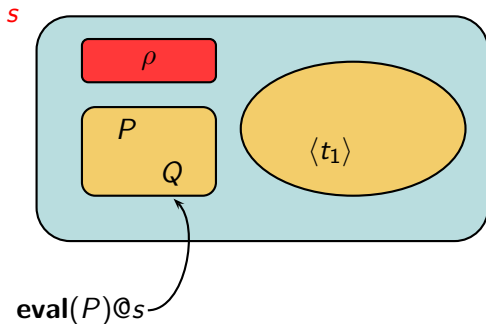
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



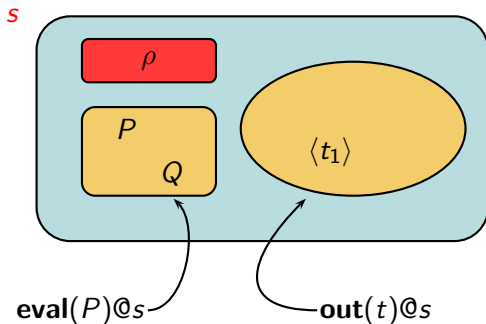
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



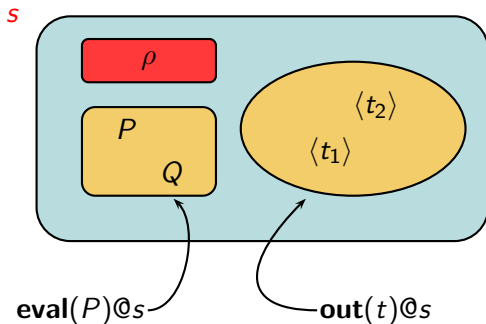
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



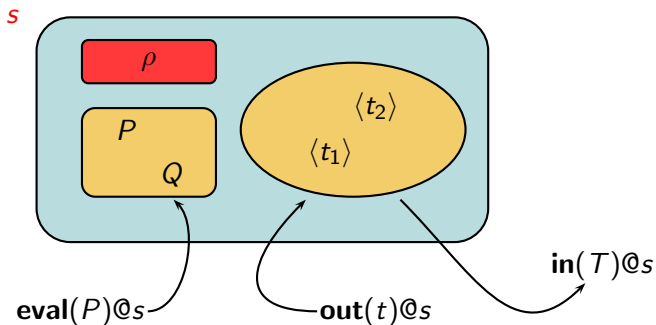
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



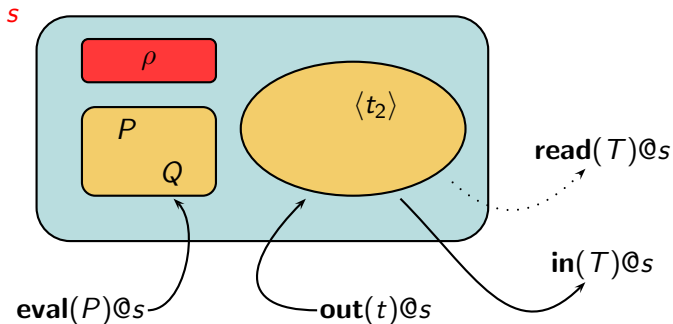
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



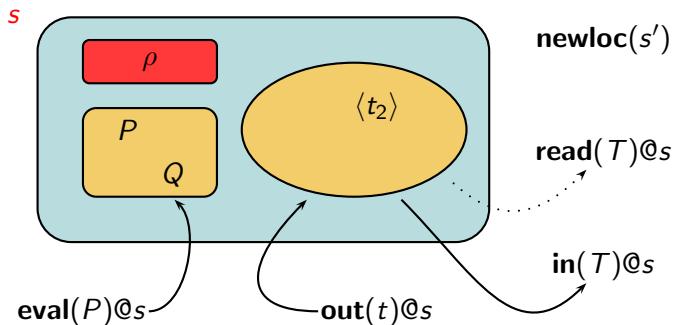
KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space

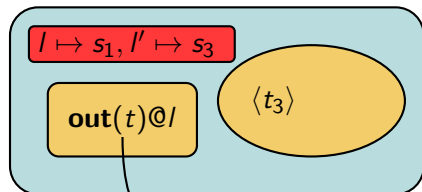
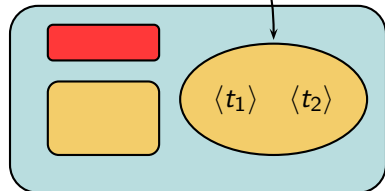
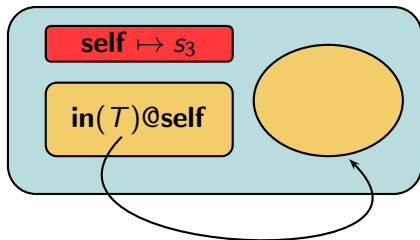


KLAIM Nodes

- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



KLAIM Nets

 s_2

 s_1

 s_3


KLAIM: Syntax (1)

Locality names: s, s', \dots

Locality variables: **self**, l, l', \dots

l will stand for a locality name or a locality variable

KLAIM: Syntax (1)

Locality names: s, s', \dots

Locality variables: **self**, l, l', \dots

l will stand for a locality name or a locality variable

Tuples:

$$t ::= e \mid \ell \mid P \mid t_1, t_2$$

Templates:

$$T ::= e \mid \ell \mid P \mid !x \mid !l \mid !X \mid T_1, T_2$$

KLAIM: Syntax (1)

Locality names: s, s', \dots

Locality variables: **self**, l, l', \dots

l will stand for a locality name or a locality variable

Tuples:

$$t ::= e \mid \ell \mid P \mid t_1, t_2$$

Templates:

$$T ::= e \mid \ell \mid P \mid !x \mid !l \mid !X \mid T_1, T_2$$

Actions:

$$a ::= \mathbf{in}(T)@l \mid \mathbf{read}(T)@l \mid \mathbf{out}(t)@l \mid \mathbf{eval}(P)@l \mid \mathbf{newloc}(s)$$

exchanging data, spawning new processes, creating new nodes

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes

Nodes have a unique address, an allocation environment and host components

$$N ::= \mathbf{0} \mid s ::_{\rho} C \mid N_1 \parallel N_2 \mid (\nu s)N$$

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes

Nodes have a unique address, an allocation environment and host components

$$N ::= \mathbf{0} \mid s ::_{\rho} C \mid N_1 \parallel N_2 \mid (\nu s)N$$

Components are evaluated tuples or running processes

$$C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$$

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes

Nodes have a unique address, an allocation environment and host components

$$N ::= \mathbf{0} \mid s ::_{\rho} C \mid N_1 \parallel N_2 \mid (\nu s)N$$

Components are evaluated tuples or running processes

$$C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$$

Processes are built up from the inert process via action prefixing, parallel composition and process invocation (and definition)

$$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid A(\bar{p}) \quad (A(\bar{f}) \triangleq P)$$

KLAIM: Syntax

Nets:

$$N ::= \mathbf{0} \mid s ::_{\rho} C \mid N_1 \parallel N_2 \mid (\nu s)N$$

Components:

$$C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$$

Processes:

$$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid A(\bar{p}) \quad (A(\bar{f}) \triangleq P)$$

Actions:

$$a ::= \mathbf{in}(T)@l \mid \mathbf{read}(T)@l \mid \mathbf{out}(t)@l \mid \mathbf{eval}(P)@l \mid \mathbf{newloc}(s)$$

Tuples:

$$t ::= e \mid l \mid P \mid t_1, t_2$$

Templates:

$$T ::= e \mid l \mid P \mid !x \mid !l \mid !X \mid T_1, T_2$$

Binders

- Prefix **in**($\dots, !x, !l, !X, \dots$)@ l . P binds x, l, X in P
(**read** acts similarly)
- Prefix **newloc**(s). P binds s in P
- Restriction $(\nu s)N$ binds s in N

KLAIM: Structural Congruence

Monoid laws for “ \parallel ”, i.e.

$$N \parallel \mathbf{0} \equiv N, N_1 \parallel N_2 \equiv N_2 \parallel N_1, (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$$

$$\text{(ALPHA)} \quad N \equiv N' \quad \text{if } N =_{\alpha} N'$$

$$\text{(RCOM)} \quad (\nu s_1)(\nu s_2)N \equiv (\nu s_2)(\nu s_1)N$$

$$\text{(EXT)} \quad N_1 \parallel (\nu s)N_2 \equiv (\nu s)(N_1 \parallel N_2) \quad \text{if } s \notin fl(N_1)$$

$$\text{(ABS)} \quad s ::_{\rho} C \equiv s ::_{\rho} (C \mid \mathbf{nil})$$

$$\text{(CLONE)} \quad s ::_{\rho} C_1 \mid C_2 \equiv s ::_{\rho} C_1 \parallel s ::_{\rho} C_2$$

$$\text{(PDEF)} \quad s ::_{\rho} A(\bar{p}) \equiv s ::_{\rho} P[\bar{p}/\bar{f}] \quad \text{if } A(\bar{f}) \triangleq P$$

KLAIM: Operational Semantics (1)

Producing data:

$$\frac{\rho(\ell) = s' \quad \mathcal{E}[\![t]\!]_{\rho} = t'}{s ::_{\rho} \mathbf{out}(t)@l.P \parallel s' ::_{\rho'} \mathbf{nil} \longmapsto s ::_{\rho} P \parallel s' ::_{\rho'} \langle t' \rangle}$$

where

- the local environment ρ is used to determine the location s' that is the target of the action
- function $\mathcal{E}[\![t]\!]_{\rho}$ replaces the free locality variables occurring in t according to the *local* environment ρ yielding t'

→ *static binding discipline*

Remark:

Free variables act as *aliases* that have to be replaced by *network addresses*

KLAIM: Operational Semantics (2)

Consuming data:

$$\frac{\rho(l) = s' \quad \text{match}(\mathcal{E}\llbracket T \rrbracket_{\rho}, t) = \sigma}{s ::_{\rho} \mathbf{in}(T)@l.P \parallel s' ::_{\rho'} \langle t \rangle \mapsto s ::_{\rho} P\sigma \parallel s' ::_{\rho'} \mathbf{nil}}$$

where:

- function $\text{match}(T, t)$ checks compliance of $\mathcal{E}\llbracket T \rrbracket_{\rho}$ and t and associates values (i.e. basic values, names and processes) to variables bound by the template
- a tuple matches against a template if they have the same number of fields, and corresponding fields match
- σ is a substitution of names and processes for the free variables in T

Matching Function

$$\text{match}(!x, v) = [v/x] \quad \text{match}(v, v) = \epsilon$$

$$\text{match}(!l, s) = [s/l] \quad \text{match}(s, s) = \epsilon$$

$$\text{match}(!X, P) = [P/X]$$

$$\frac{\text{match}(T_1, t_1) = \sigma_1 \quad \text{match}(T_2, t_2) = \sigma_2}{\text{match}((T_1, T_2), (t_1, t_2)) = \sigma_1 \circ \sigma_2}$$

KLAIM: Operational Semantics (3)

Accessing data:

$$\frac{\rho(l) = s' \quad \text{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{s ::_{\rho} \mathbf{read}(T)@l.P \parallel s' ::_{\rho'} \langle t \rangle \longmapsto s ::_{\rho} P\sigma \parallel s' ::_{\rho'} \langle t \rangle}$$

read is similar to **in**, but the accessed tuple is left in the tuple space

KLAIM: Operational Semantics (4)

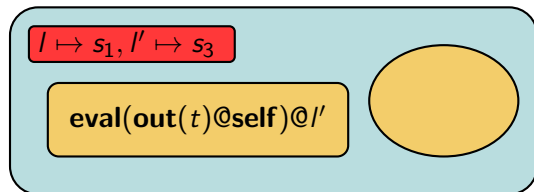
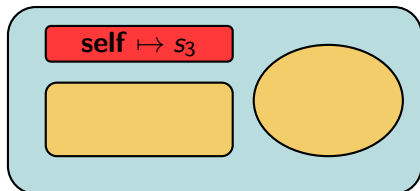
Spawning new processes:

$$\frac{\rho(\ell) = s'}{s ::_{\rho} \mathbf{eval}(Q)@l.P \parallel s' ::_{\rho'} \mathbf{nil} \mapsto s ::_{\rho} P \parallel s' ::_{\rho'} Q}$$

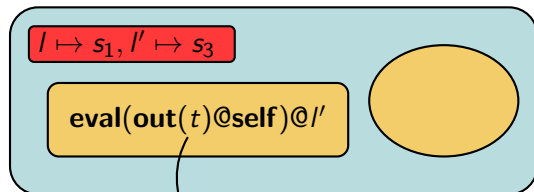
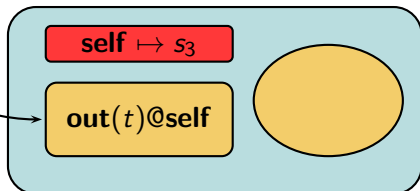
No interpretation of **eval**'s argument is performed: free variables in Q will be translated according to the *remote* allocation environment

→ *dynamic binding discipline*

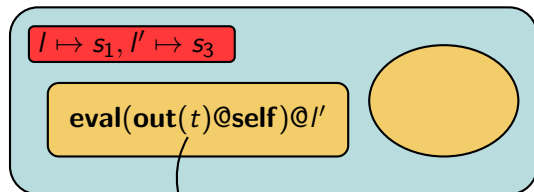
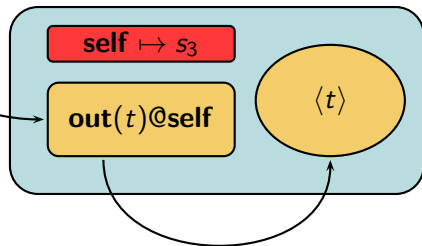
Dynamic Scoping

 s_2  s_3 

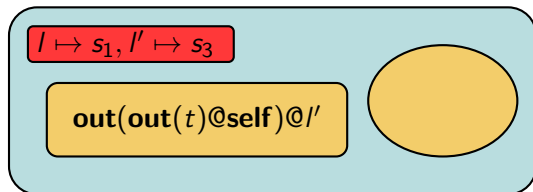
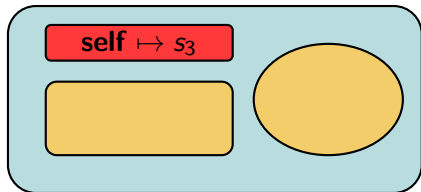
Dynamic Scoping

 s_2  s_3 

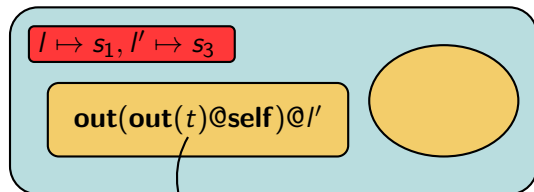
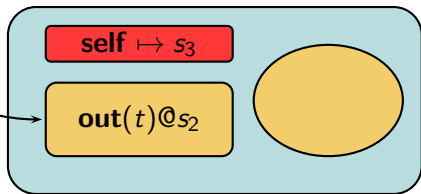
Dynamic Scoping

 s_2  s_3 

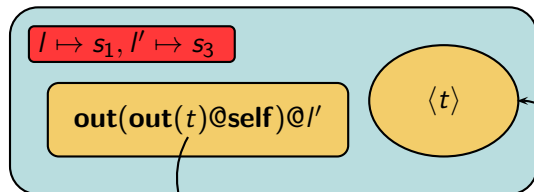
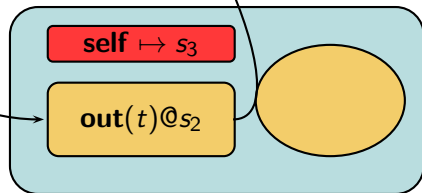
Static Scoping

 s_2  s_3 

Static Scoping

 s_2  s_3 

Static Scoping

 s_2  s_3 

KLAIM: Operational Semantics (5)

Creating new nodes:

$$s ::_{\rho} \mathbf{newloc}(s').P \mapsto (\nu s')(s ::_{\rho} P \parallel s' ::_{\rho[s'/\mathbf{self}]} \mathbf{nil}) \quad \text{with } s' \notin \text{codom}(\rho)$$

where the environment of the creator is used to build up the environment for the new node

KLAIM: Operational Semantics (7)

Rules summary:

$$\frac{\rho(\ell) = s' \quad \mathcal{E}[\![t]\!]_{\rho} = t'}{s ::_{\rho} \mathbf{out}(t)@l.P \parallel s' ::_{\rho'} \mathbf{nil} \mapsto s ::_{\rho} P \parallel s' ::_{\rho'} \langle t' \rangle}$$

$$\frac{\rho(\ell) = s'}{s ::_{\rho} \mathbf{eval}(Q)@l.P \parallel s' ::_{\rho'} \mathbf{nil} \mapsto s ::_{\rho} P \parallel s' ::_{\rho'} Q}$$

$$\frac{\rho(\ell) = s' \quad \mathit{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{s ::_{\rho} \mathbf{in}(T)@l.P \parallel s' ::_{\rho'} \langle t \rangle \mapsto s ::_{\rho} P\sigma \parallel s' ::_{\rho'} \mathbf{nil}}$$

$$\frac{\rho(\ell) = s' \quad \mathit{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{s ::_{\rho} \mathbf{read}(T)@l.P \parallel s' ::_{\rho'} \langle t \rangle \mapsto s ::_{\rho} P\sigma \parallel s' ::_{\rho'} \langle t \rangle}$$

$$s ::_{\rho} \mathbf{newloc}(s').P \mapsto (\nu s')(s ::_{\rho} P \parallel s' ::_{\rho[s'/\mathbf{self}]} \mathbf{nil}) \quad \text{with } s' \notin \text{codom}(\rho)$$

KLAIM: Operational Semantics (6)

Remaining rules:

$$\frac{N_1 \mapsto N'_1}{N_1 \parallel N_2 \mapsto N'_1 \parallel N_2}$$

$$\frac{N \mapsto N'}{(\nu s)N \mapsto (\nu s)N'}$$

$$\frac{N \equiv M \mapsto M' \equiv N'}{N \mapsto N'}$$

Leader Election (assumptions)

- n participants distributed on the nodes of a network must elect a *leader*
 - Each participant is univocally identified by an *id*
 - The leader will be the participant with the smaller *id*
- The topology of the network is a ring:
every node is directly connected to two other nodes (*prev* and *next*)
 - n nodes with addresses s_0, \dots, s_{n-1}
 - The allocation environment ρ_i of node s_i , in addition to the standard mapping **self** $\mapsto s_i$, maps the logical locality *next* to $s_{i+1 \bmod n}$
- At the outset, each process retrieves an *id* from the node *rg* that acts as a random generator
 - The *ids* are all different

Leader Election: algorithm

- Once a participant has assigned an *id*, it spawns the mobile process *checker* to the next node
 - This process travels along the ring to determine if the source node has to be the leader
- *checker* carries the *id* of the source node, retrieves the *id* of the node where it is running (stored in x) and compares the two *ids*
 - If $id < x$, the currently hosting node is not the leader: the process moves to the next node and restarts
 - If $id > x$, the source node is not the leader: the process crosses the rest of the ring to come back to the source node and insert this information in the local tuple space
 - Otherwise, i.e. $id = x$, the process is back on the source node (thus no node with a smaller *id* has been found in the ring) and inserts the information that this is the leader in the local tuple space

Leader Election: specification

$P \triangleq$ **in**("ID", ! *id*)@*rg*.
out("ID", *id*)@**self**.
eval(*checker*(*id*))@*next*.**nil**

checker(*myld*) \triangleq **read**("ID", ! *x*)@**self**.
if *myld* < *x* **then**
 eval(*checker*(*myld*))@*next*.**nil**
 else if *myld* > *x* **then**
 eval(*notifier*(*myld*))@*next*.**nil**
 else out("LEADER")@**self**.**nil**

notifier(*myld*) \triangleq **read**("ID", ! *x*)@**self**.
if *x* = *myld* **then**
 out("FOLLOWER")@**self**.**nil**
 else eval(*notifier*(*myld*))@*next*.**nil**

(Mild) Extensions

By exploiting pattern matching and parallel composition, the conditional construct

```

if bcond
  then P
  else Q

```

can be rendered as

```

out("EVALUATE", bcond)@self.
  (in("EVALUATE", true)@self.P
   |
   in("EVALUATE", false)@self.Q)

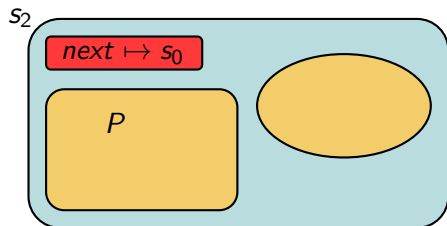
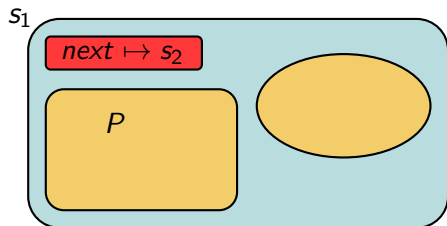
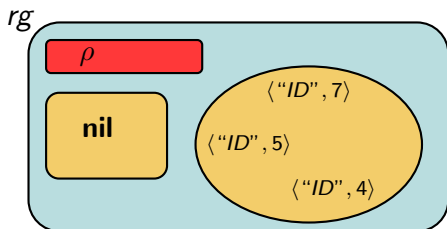
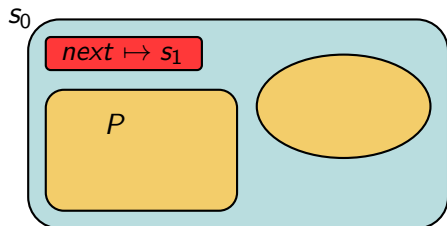
```

since only one of the two parallel processes can proceed

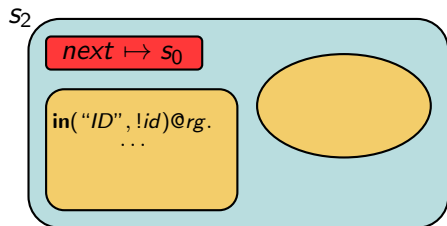
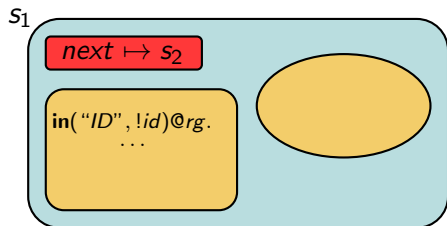
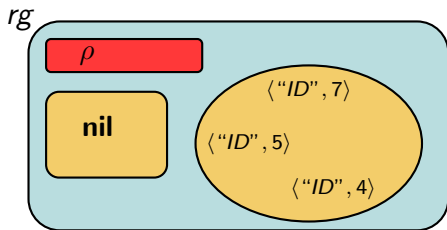
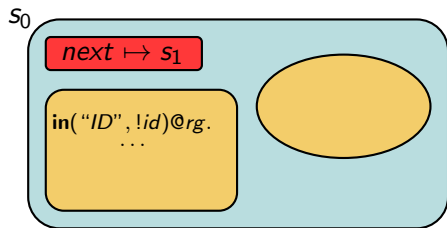
Leader Election: specification

$$\begin{aligned}
 s_0 &:: [\text{self} \mapsto s_0, \text{next} \mapsto s_1] \ P \\
 &\parallel \\
 s_1 &:: [\text{self} \mapsto s_1, \text{next} \mapsto s_2] \ P \\
 &\parallel \\
 s_2 &:: [\text{self} \mapsto s_2, \text{next} \mapsto s_0] \ P \\
 &\parallel \\
 rg &:: [\text{self} \mapsto rg] \ \langle \text{"ID"}, 7 \rangle \mid \langle \text{"ID"}, 4 \rangle \mid \langle \text{"ID"}, 5 \rangle \mid \mathbf{nil}
 \end{aligned}$$

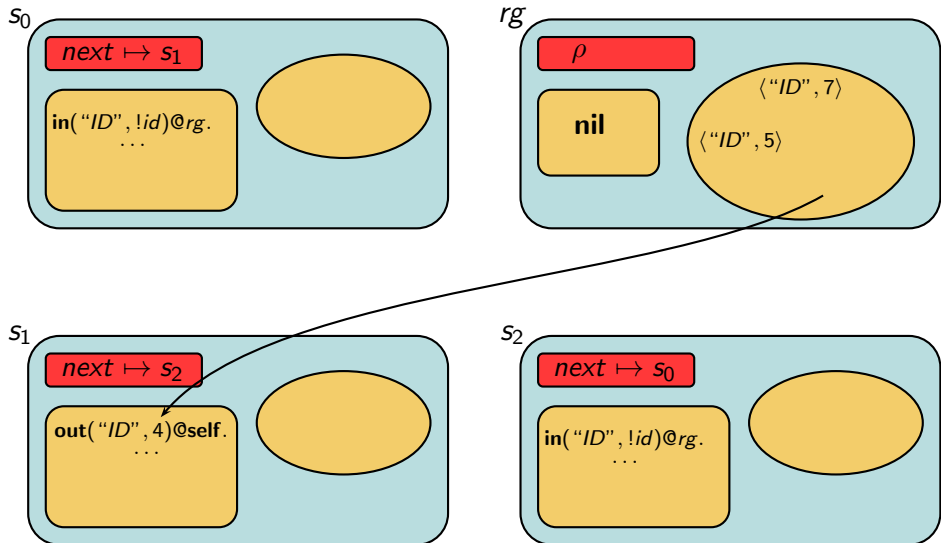
The overall system



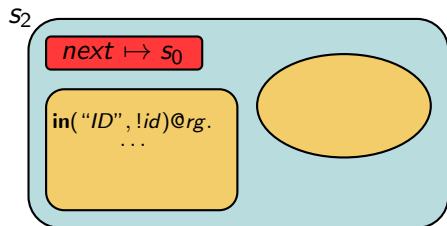
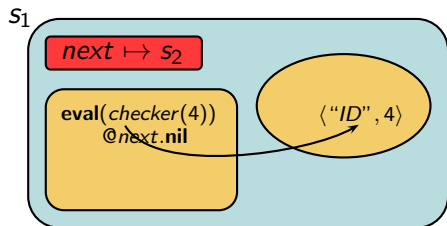
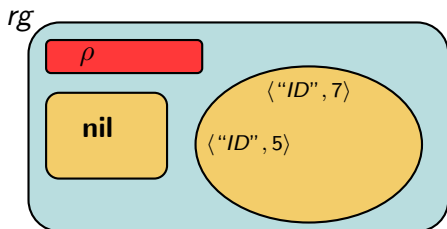
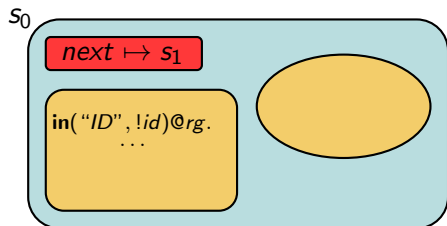
A computation



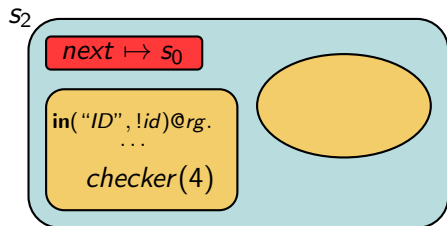
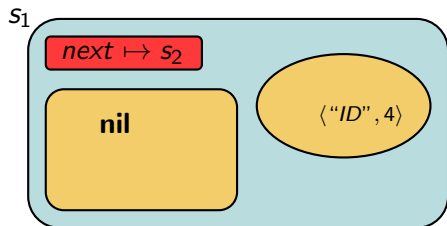
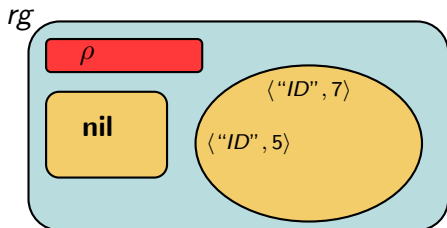
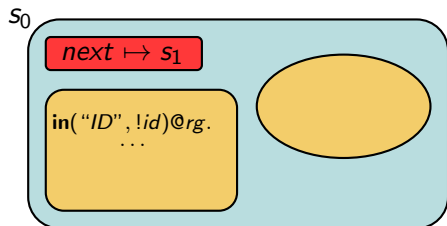
A computation



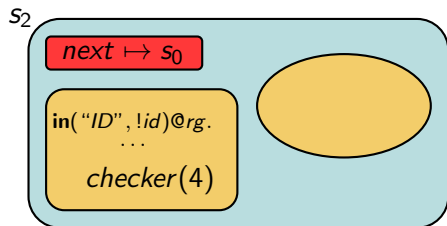
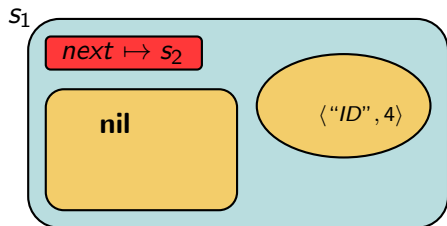
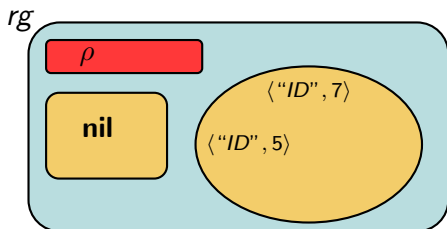
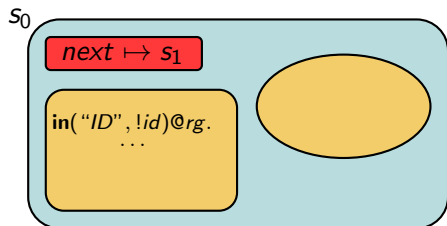
A computation



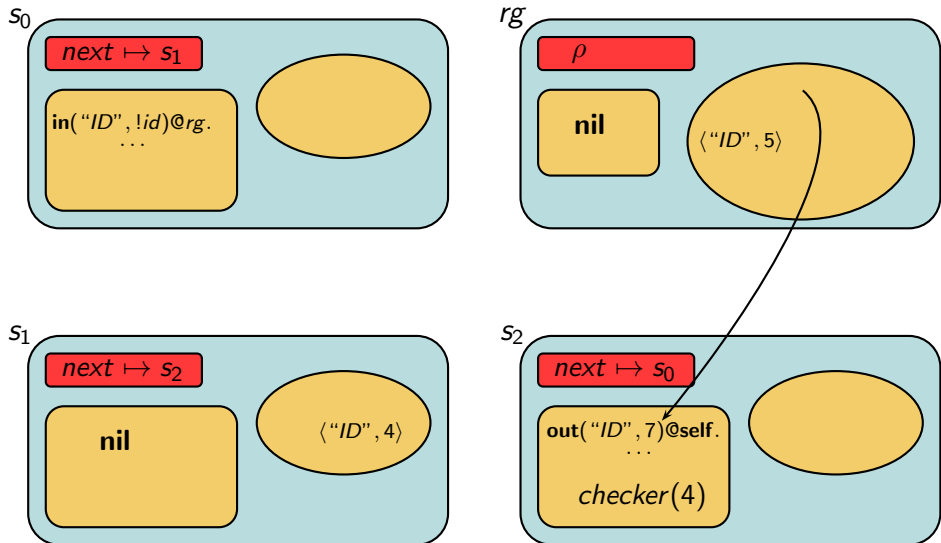
A computation



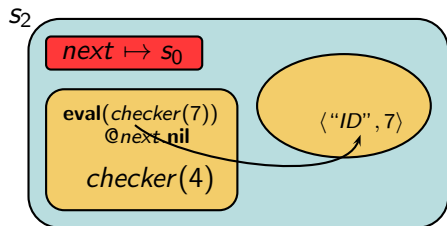
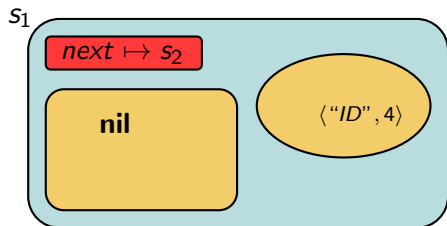
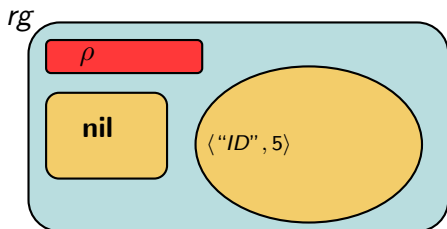
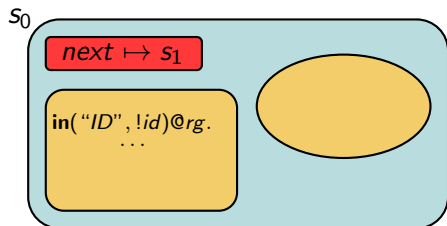
A computation



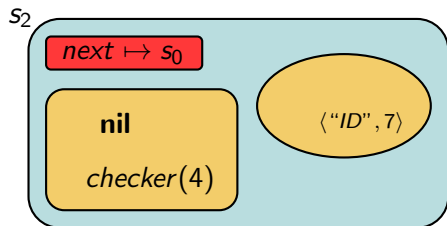
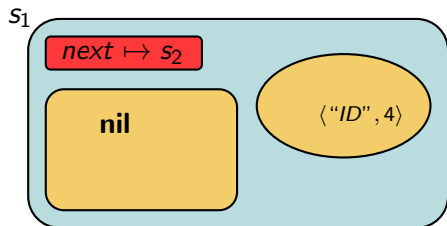
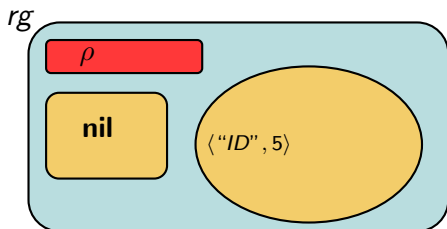
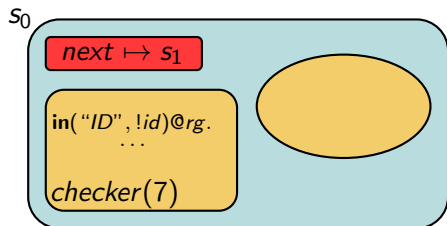
A computation



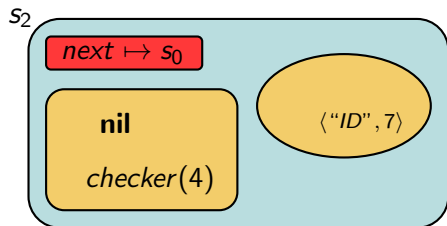
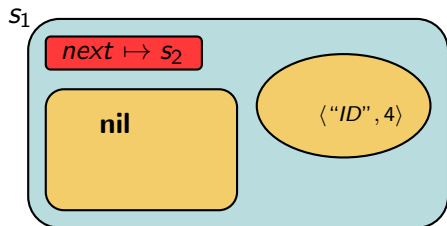
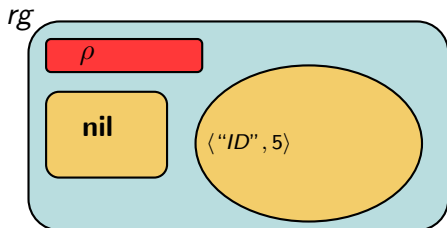
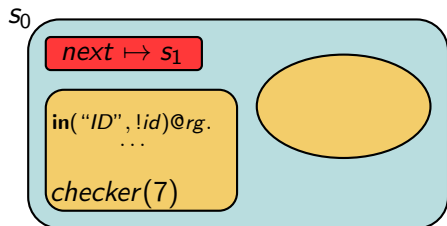
A computation



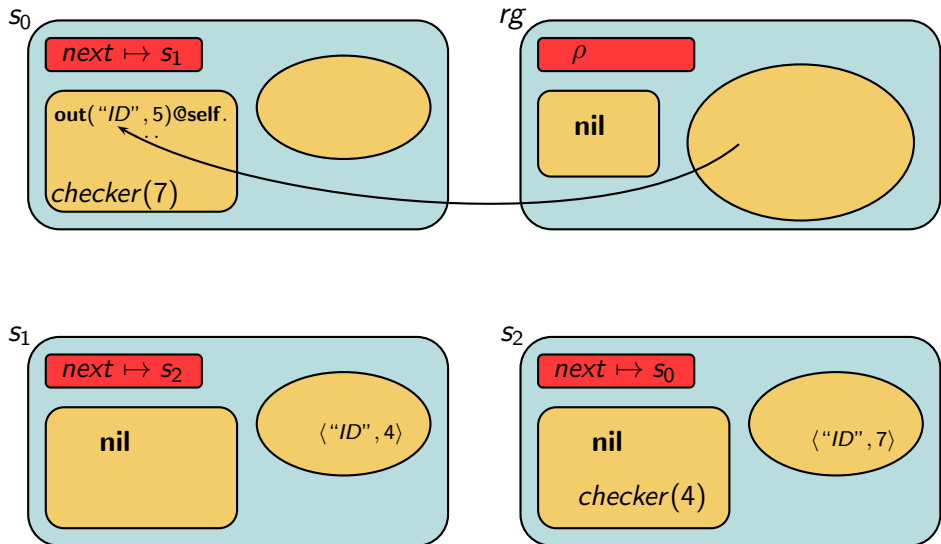
A computation



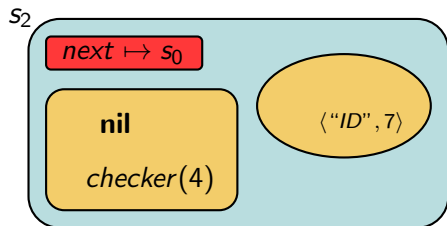
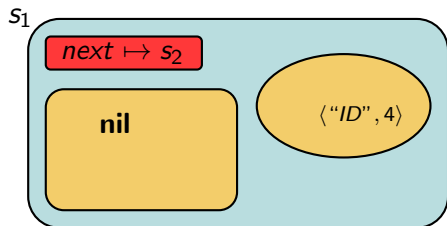
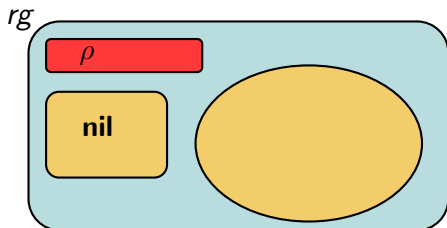
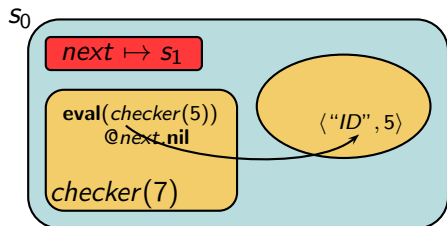
A computation



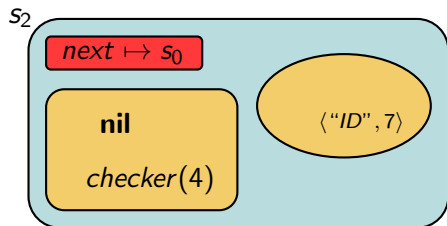
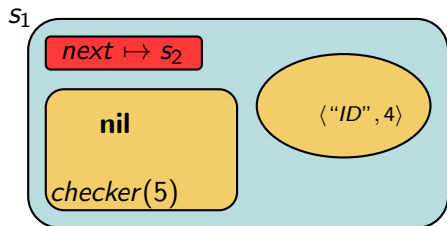
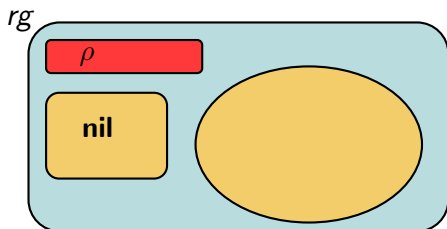
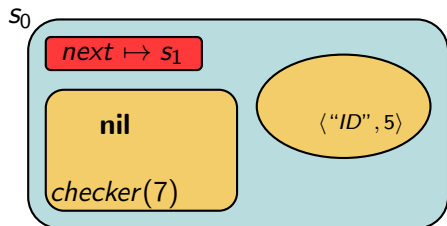
A computation



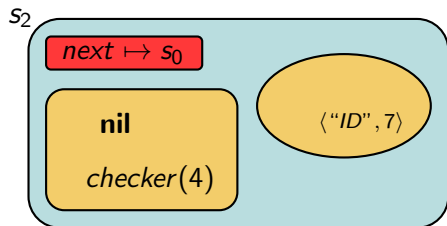
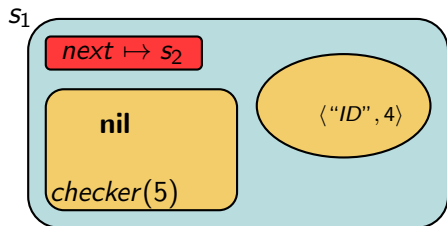
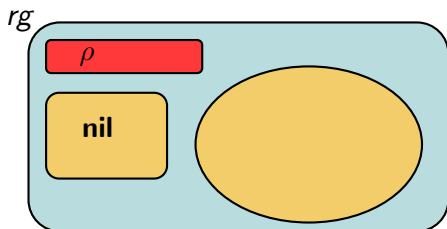
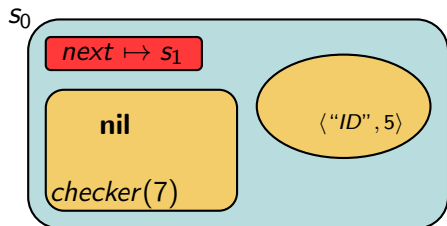
A computation



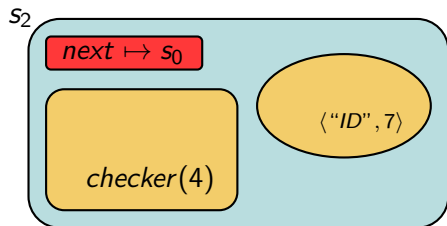
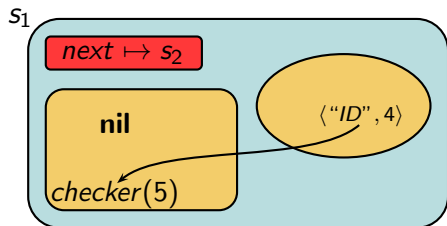
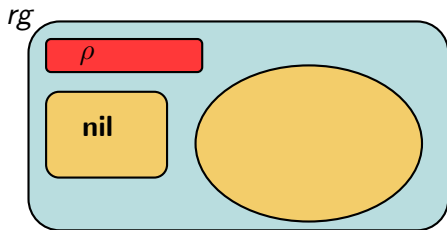
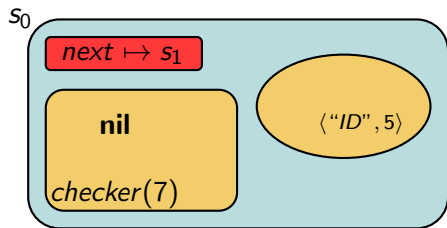
A computation



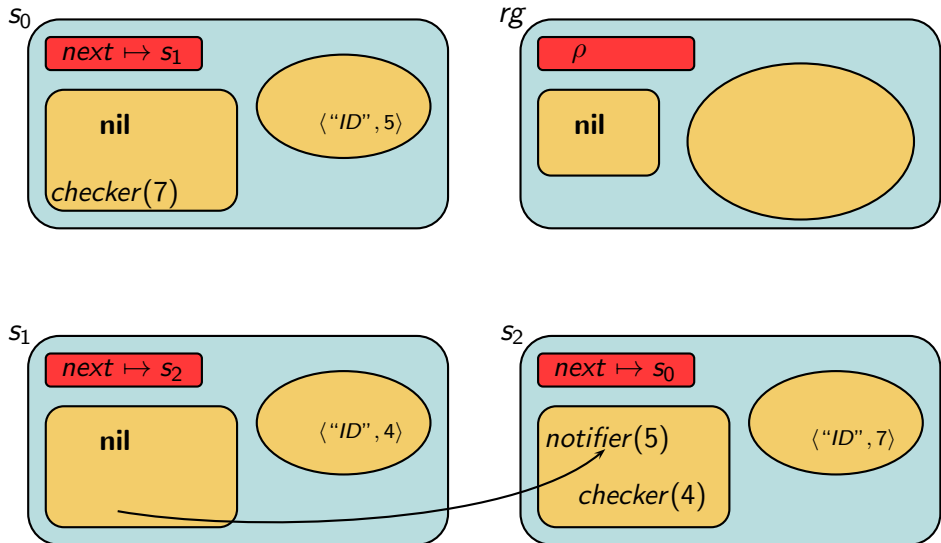
A computation



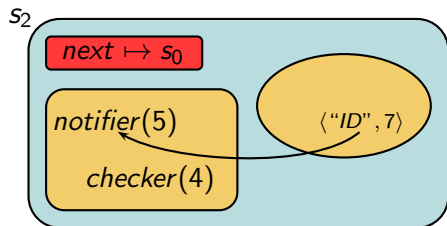
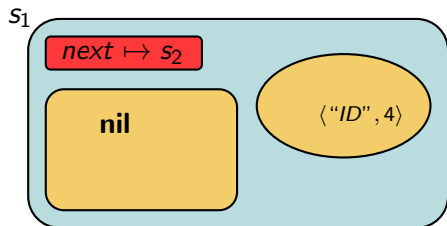
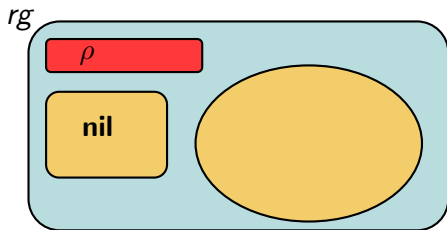
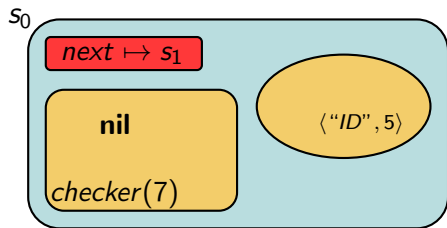
A computation



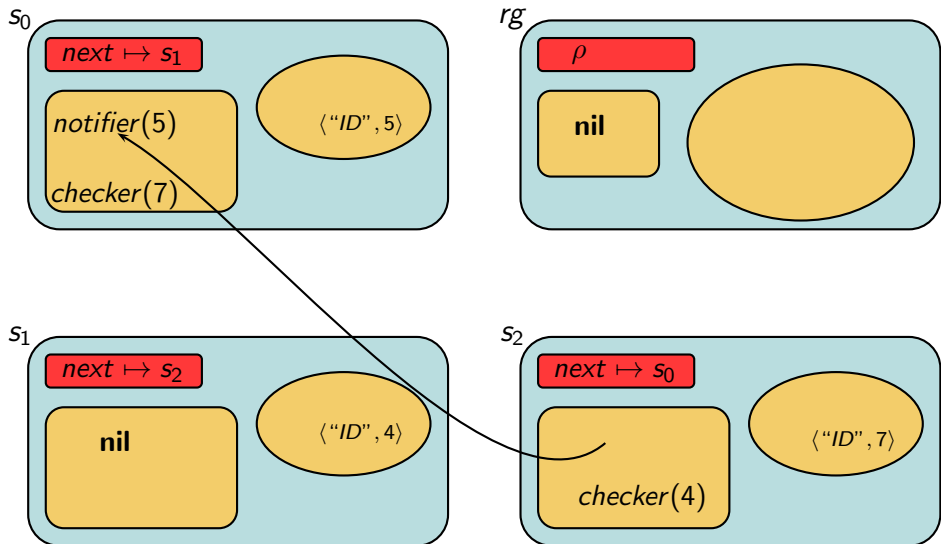
A computation



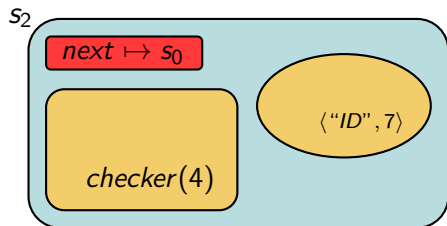
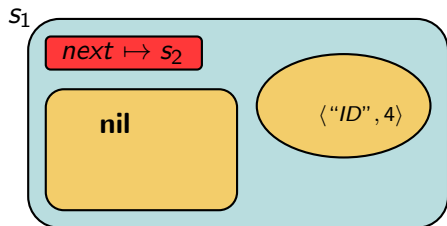
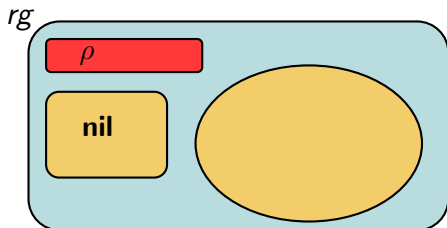
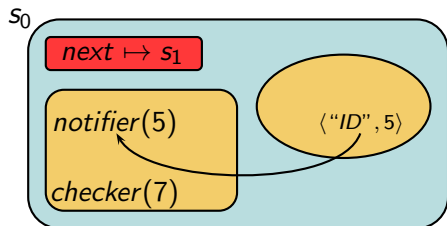
A computation



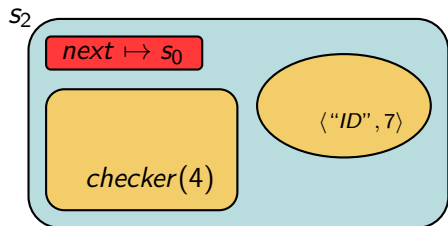
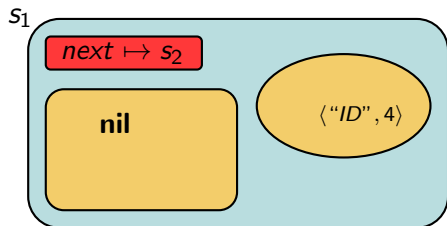
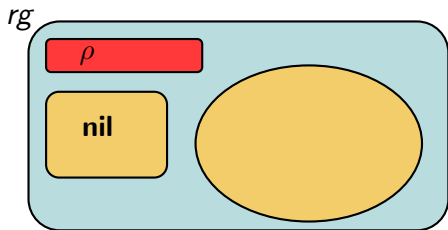
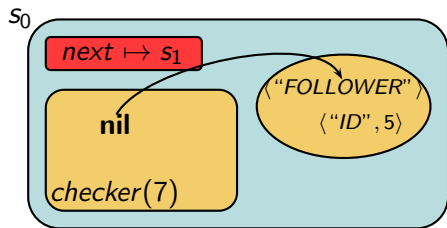
A computation



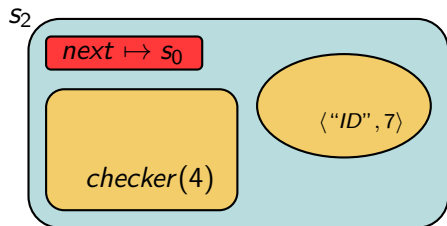
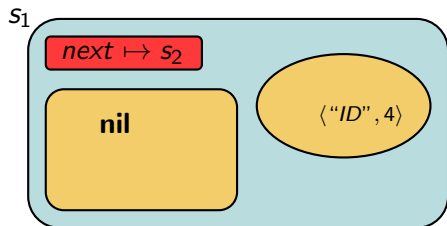
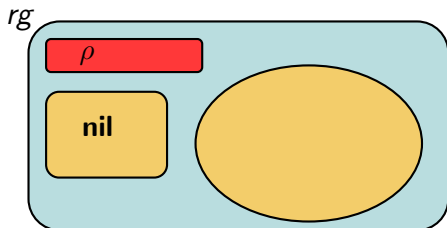
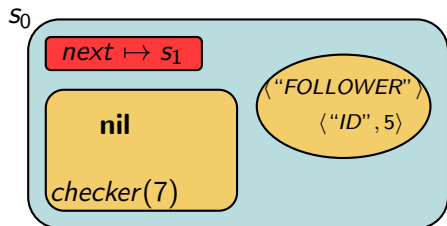
A computation



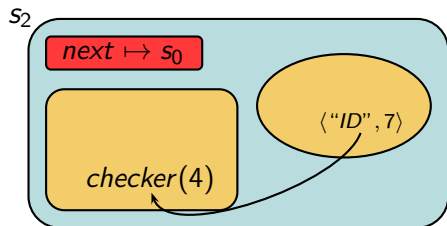
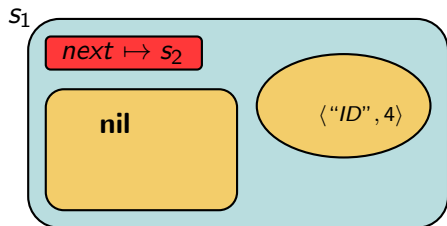
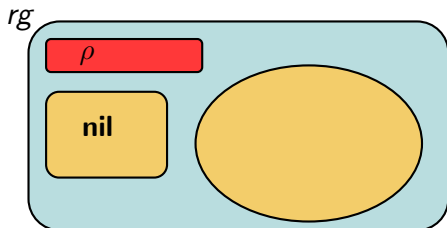
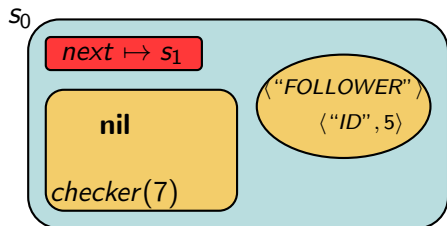
A computation



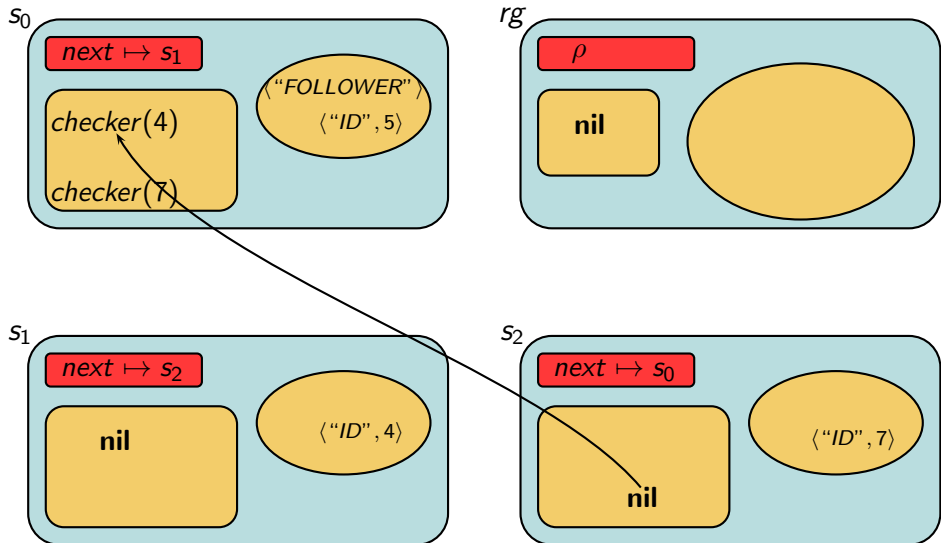
A computation



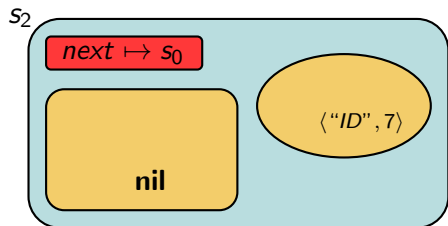
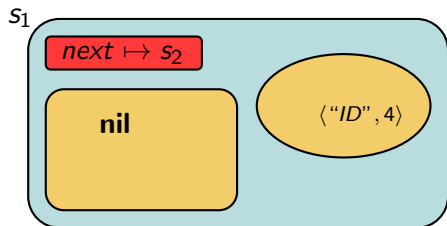
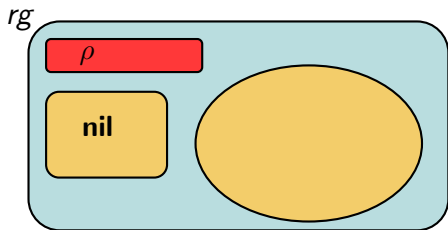
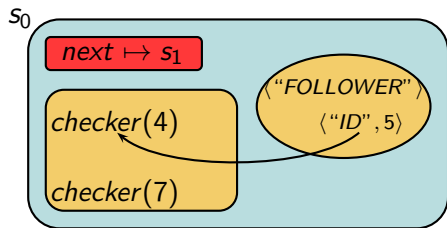
A computation



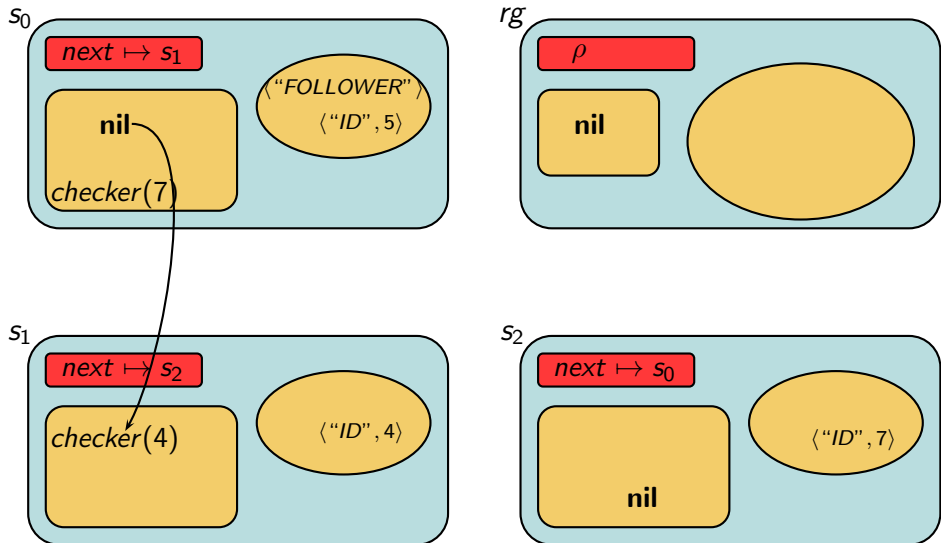
A computation



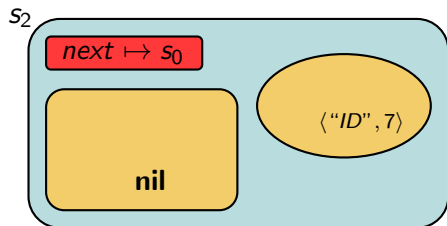
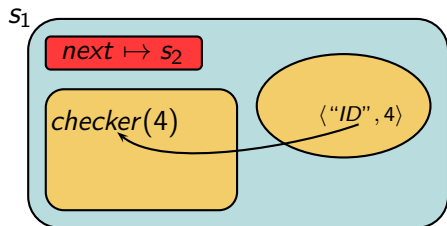
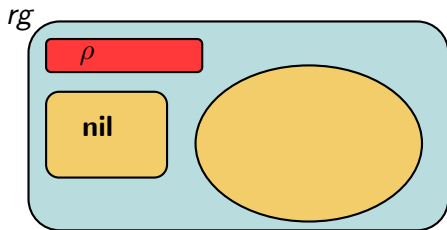
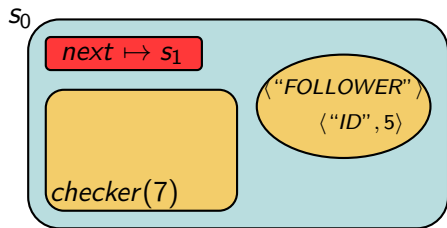
A computation



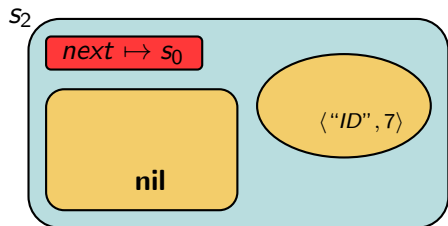
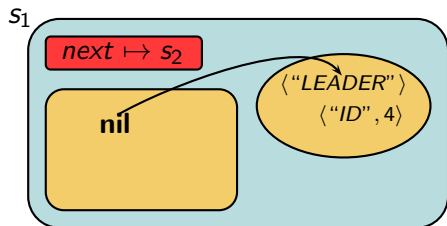
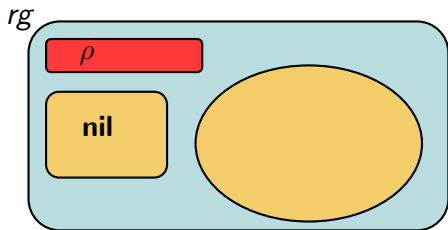
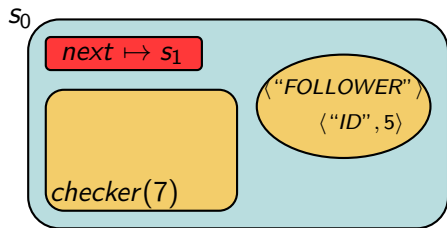
A computation



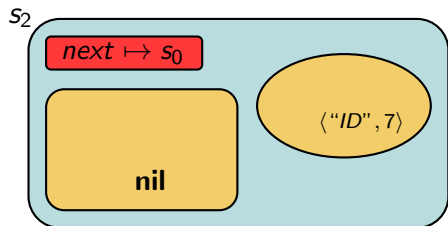
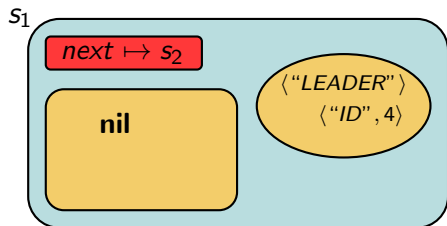
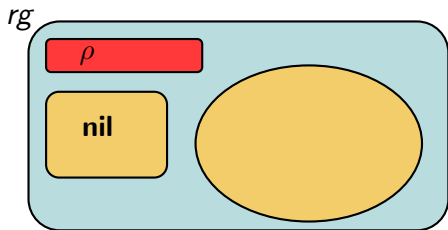
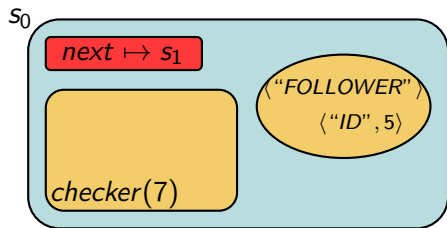
A computation



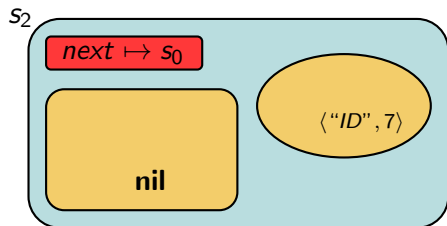
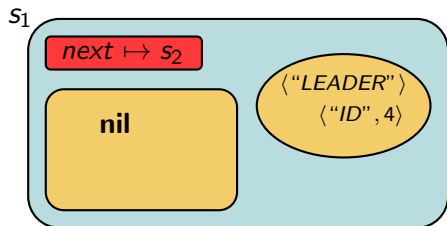
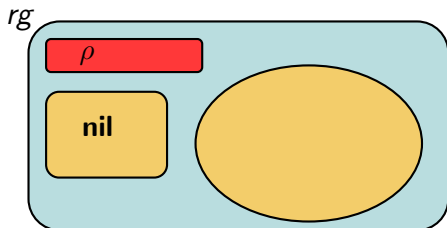
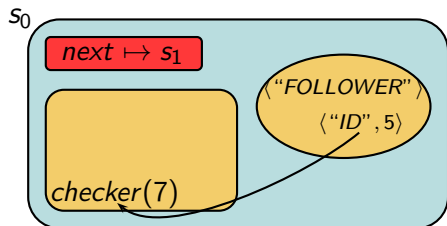
A computation



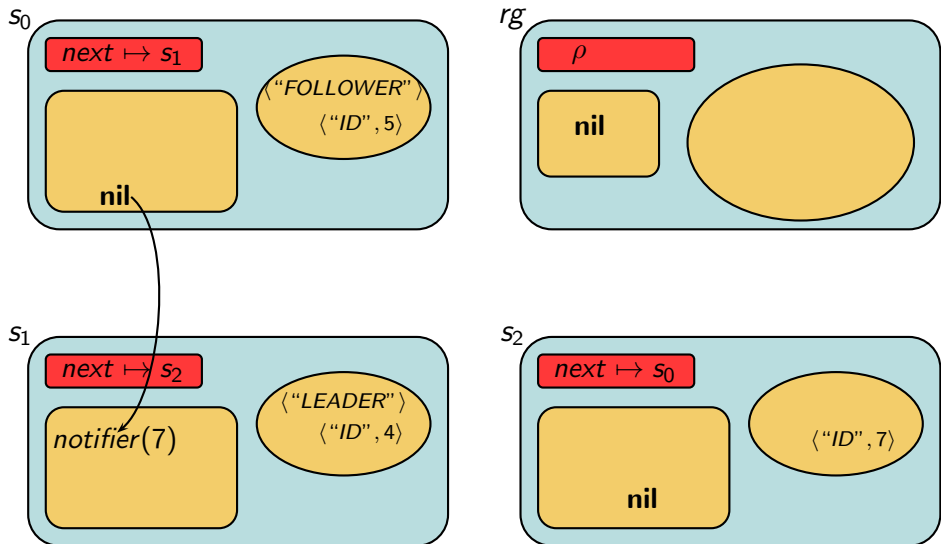
A computation



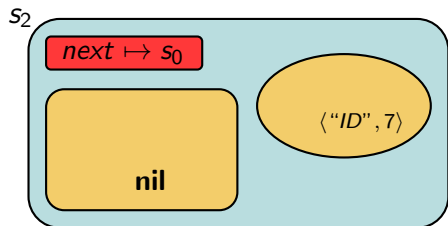
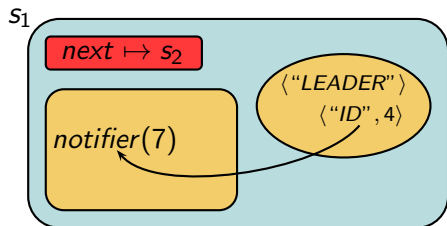
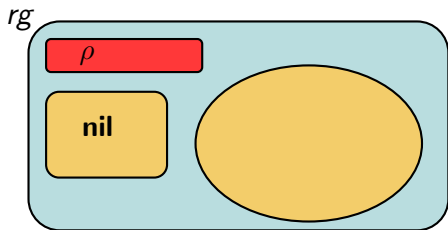
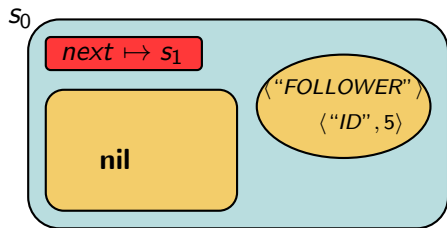
A computation



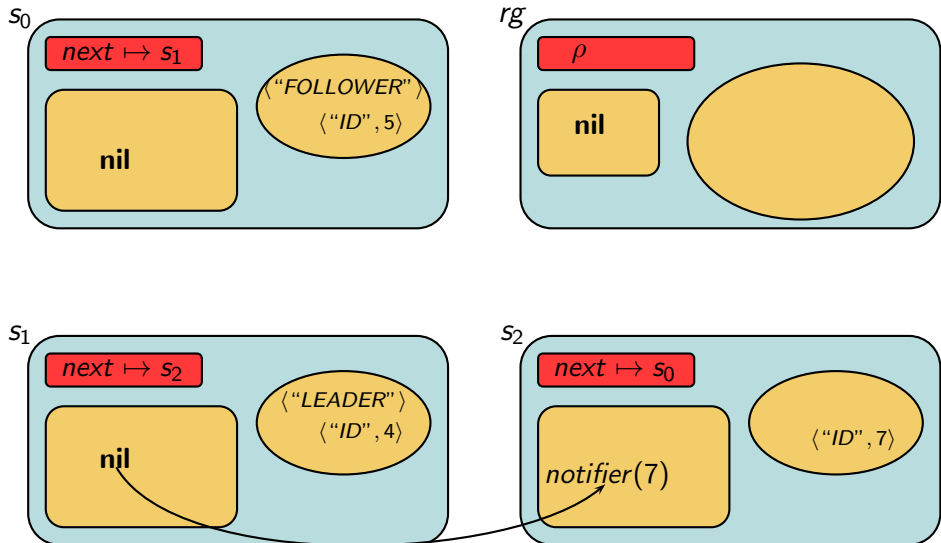
A computation



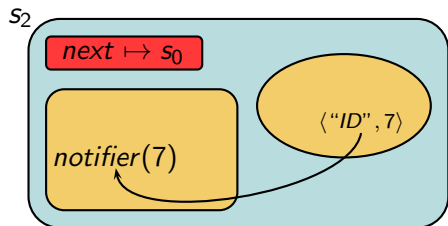
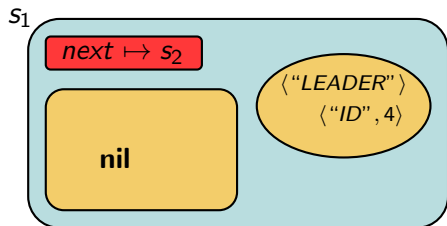
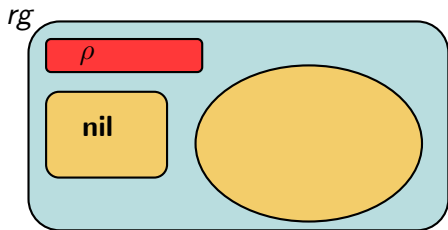
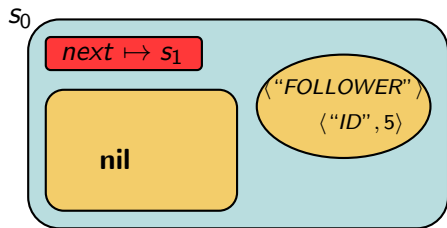
A computation



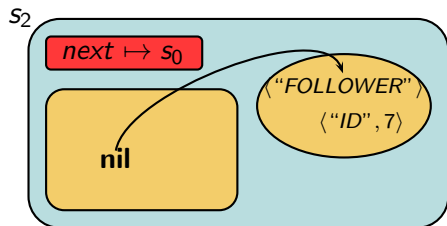
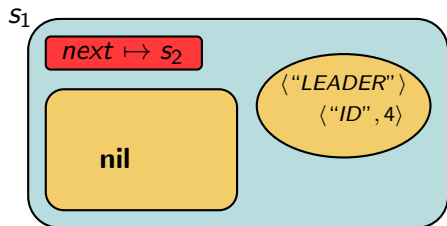
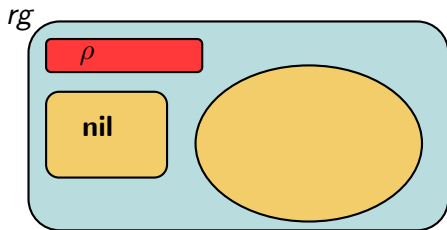
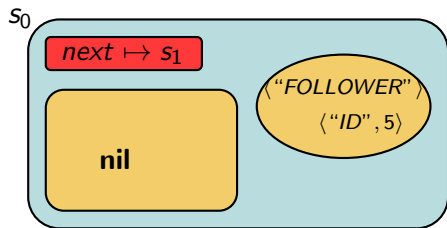
A computation



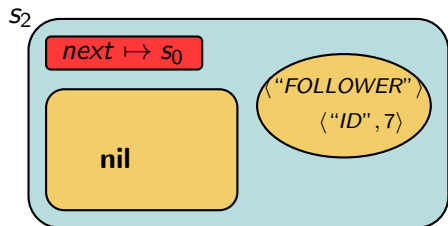
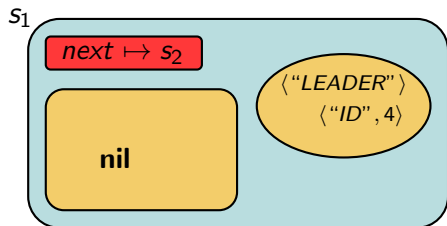
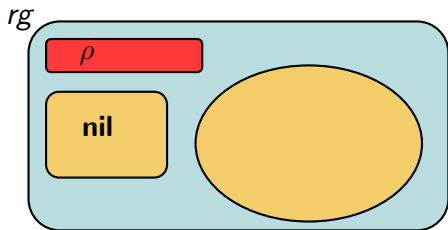
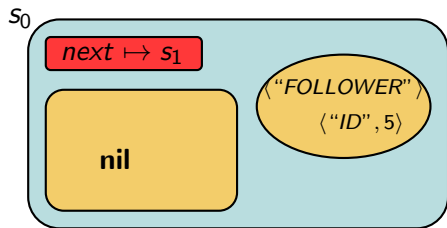
A computation



A computation



A computation



Implementation

X-KLAIM (eXtended KLAIM)

A fully-fledged programming language including the KLAIM operations and a high-level syntax:

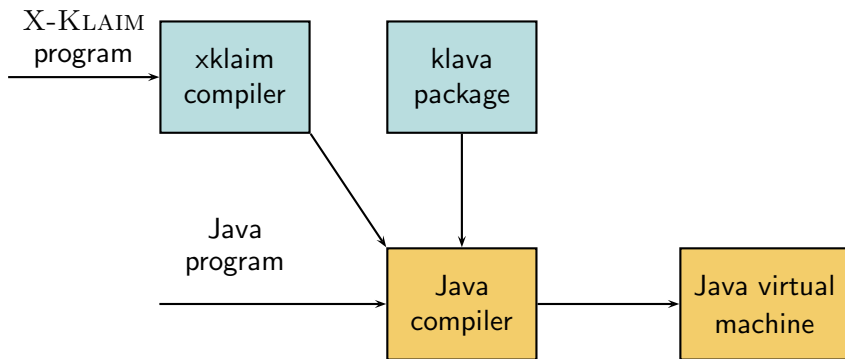
- Variable declarations
- Assignments
- Conditionals, iteration, ...
- Time-outs

Klava (KLAIM in Java)

A Java package implementing

- KLAIM operations and concepts
- Communications among processes and nodes
- Code mobility

KLAIM Programming Framework



KLAIM website:

```
http://music.dsi.unifi.it/klaim.html
```