Domain Specific Formal Languages

Klaim

Francesco Tiezzi

University of Camerino
francesco.tiezzi@unicam.it

A.A. 2017/2018

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

Opended, 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: $(\text{“}foo\text{”}, 10 + 5, true)$
  - 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: $(\text{"}\mathit{foo}\text{"}, 10 + 5, \mathit{true})$
  - 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

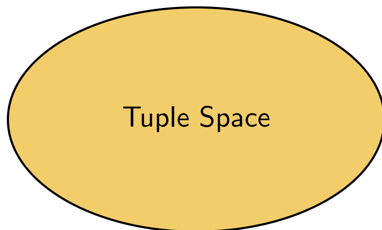
Tuples and Pattern Matching

- Tuples: $(\text{``}\mathit{foo}\text{''}, 10 + 5, \mathit{true})$
  - 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

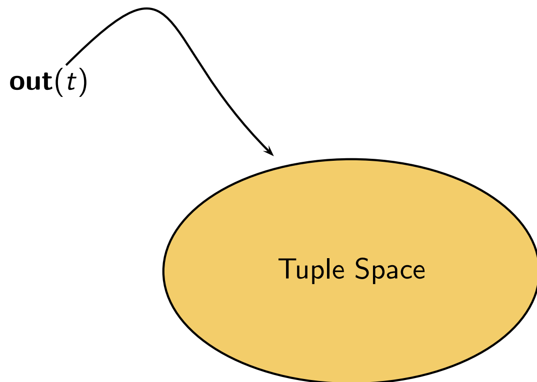
$(!s, 15, !b)$ matches $(\text{``}\mathit{foo}\text{''}, 15, \mathit{true})$

$(!s, 15, !b)$ does not match $(\text{``}\mathit{foo}\text{''}, 10, \mathit{true})$

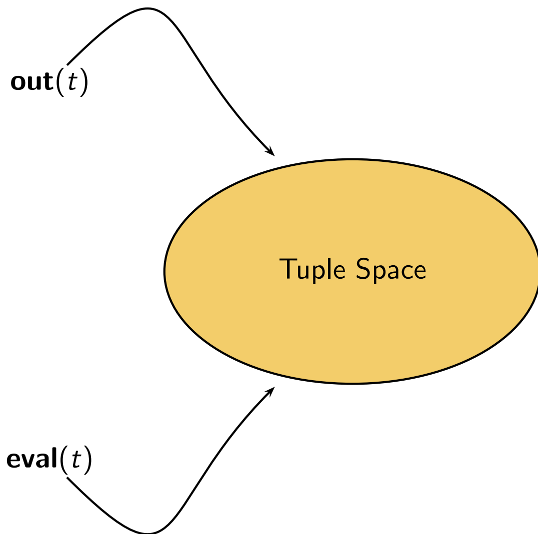
Linda communication model



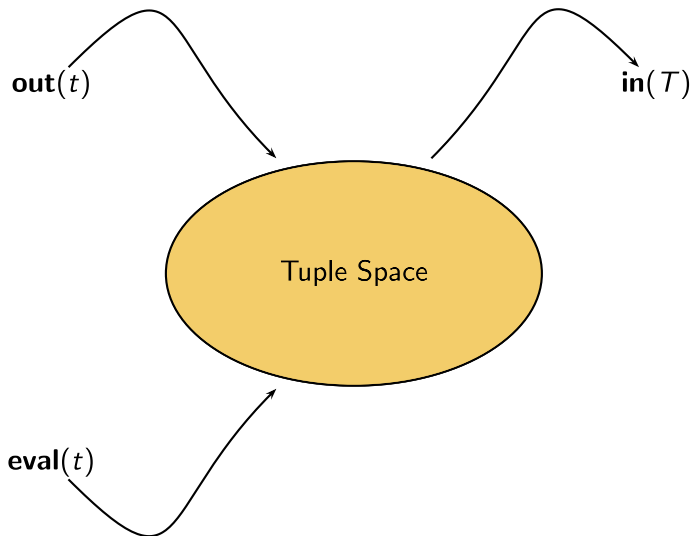
Linda communication model



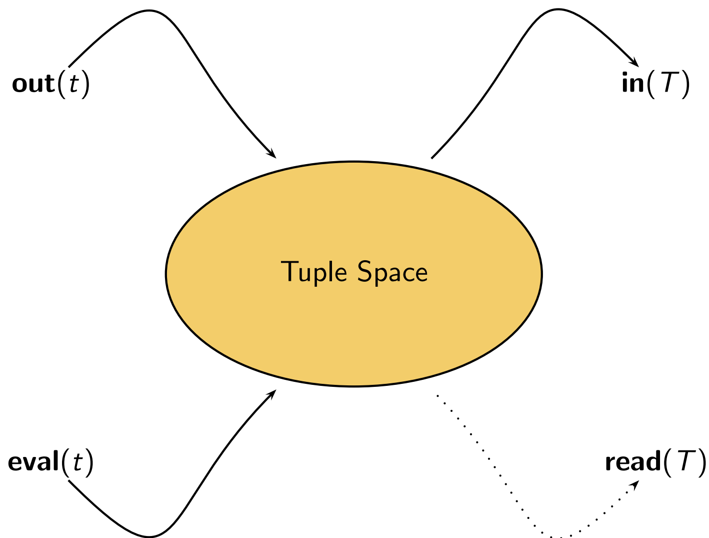
Linda communication model



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

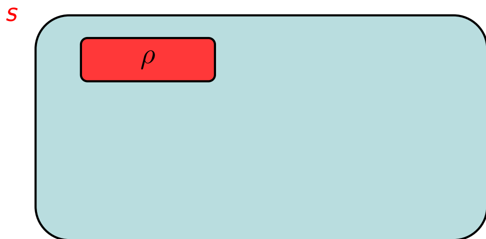
- Locality name (address)
- Allocation environment
- Processes
- Tuple Space

KLAIM Nodes

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

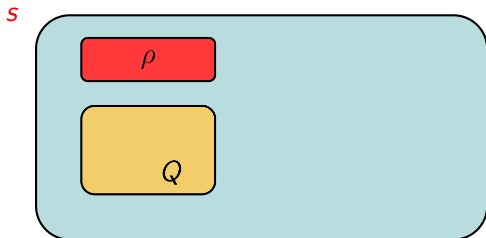
KLAIM Nodes

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



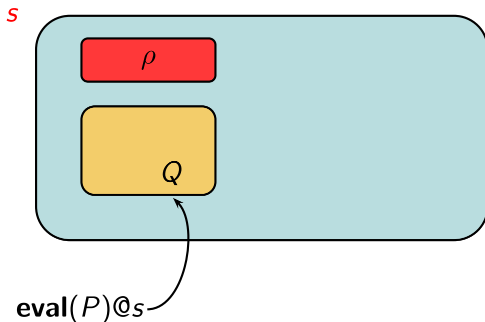
KLAIM Nodes

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



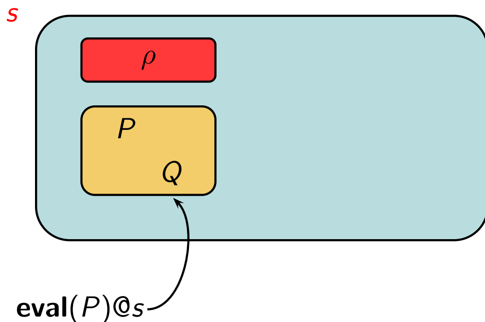
KLAIM Nodes

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



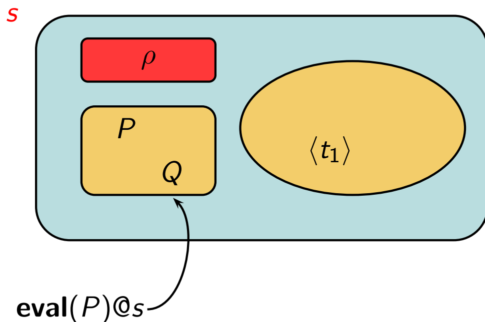
KLAIM Nodes

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



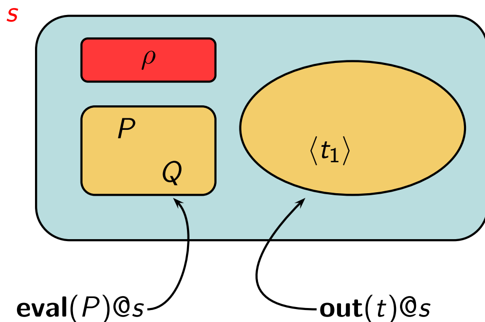
KLAIM Nodes

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



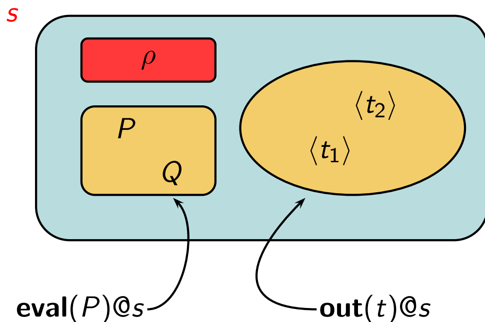
KLAIM Nodes

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



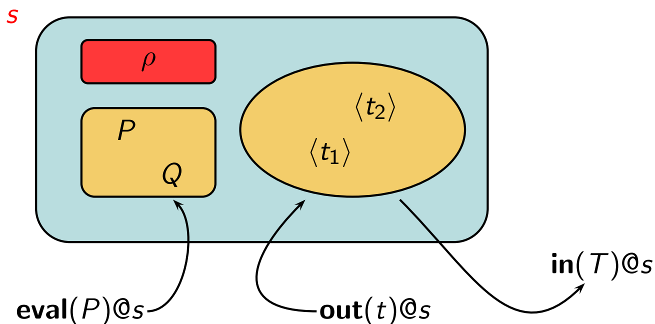
Klaim Nodes

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



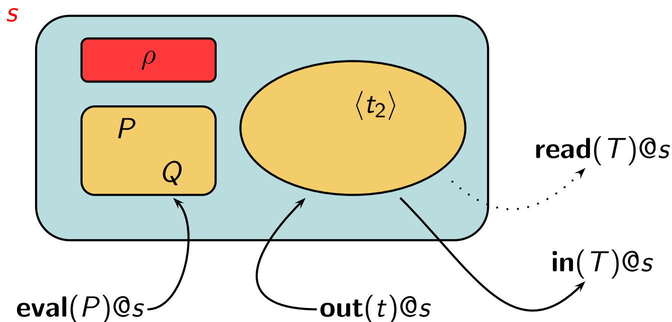
KLAIM Nodes

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



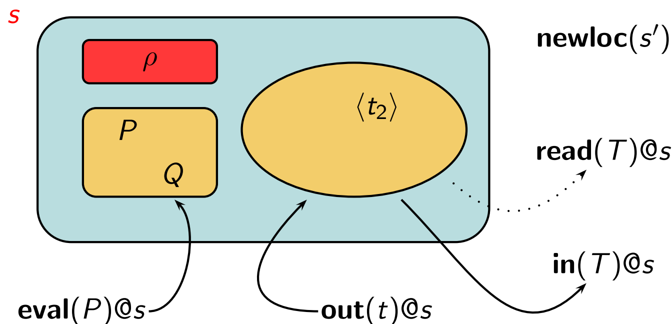
Klaim Nodes

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

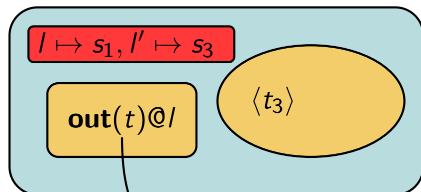


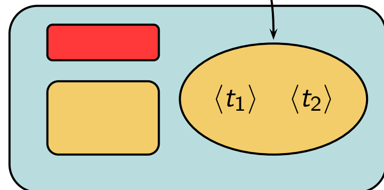
KLAIM Nodes

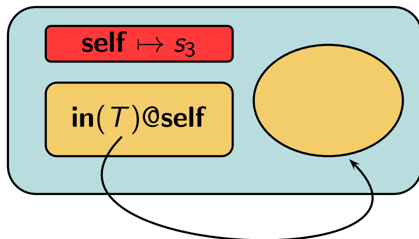
- Locality name (address)
- Allocation environment
- Processes
- Tuple Space



Klaim Nets







Klaim: Syntax (1)

Locality names: $s, s', \ldots$ *Locality variables*: $\textbf{self}, l, l', \ldots$

ℓ will stand for a locality name or a locality variable

KLAIM: Syntax (1)

Locality names: $s, s', \ldots$ *Locality variables*: $\mathbf{self}, l, l', \ldots$

ℓ will stand for a locality name or a locality variable

Tuples:

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

Templates:

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

KLAIM: Syntax (1)

Locality names: $s, s', \ldots$ *Locality variables*: $\mathbf{self}, l, l', \ldots$

ℓ will stand for a locality name or a locality variable

Tuples:

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

Templates:

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

Actions:

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

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

$$N \quad ::= \quad \mathbf{0} \;\big|\; s ::_{\rho} C \;\big|\; N_1 \parallel N_2 \;\big|\; (\nu s)N$$

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes
Nodes have a unique address, an allocation environment and host components

$$N \quad ::= \quad \mathbf{0} \;\big|\; s ::_{\rho} C \;\big|\; N_1 \parallel N_2 \;\big|\; (\nu s)N$$

Components are evaluated tuples or running processes

$$C \quad ::= \quad \langle t \rangle \;\big|\; P \;\big|\; C_1 \mid C_2$$

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes
Nodes have a unique address, an allocation environment and host components

$$N \quad ::= \quad \mathbf{0} \;\big|\; s ::_\rho C \;\big|\; N_1 \parallel N_2 \;\big|\; (\nu s)N$$

Components are evaluated tuples or running processes

$$C \quad ::= \quad \langle t \rangle \;\big|\; P \;\big|\; C_1 \mid C_2$$

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

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

KLAIM: Syntax

Nets:

$$N \quad ::= \quad \mathbf{0} \;\big|\; s ::_\rho C \;\big|\; N_1 \parallel N_2 \;\big|\; (\nu s)N$$

Components:

$$C \quad ::= \quad \langle t \rangle \;\big|\; P \;\big|\; C_1 \mid C_2$$

Processes:

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

Actions:

$$a \;::=\; \mathbf{in}(T)@\ell \;\big|\; \mathbf{read}(T)@\ell \;\big|\; \mathbf{out}(t)@\ell \;\big|\; \mathbf{eval}(P)@\ell \;\big|\; \mathbf{newloc}(s)$$

Tuples: $t \quad ::= \quad e \;\big|\; \ell \;\big|\; P \;\big|\; t_1, t_2$

Templates: $T \quad ::= \quad e \;\big|\; \ell \;\big|\; P \;\big|\; !\,x \;\big|\; !\,l \;\big|\; !\,X \;\big|\; T_1, T_2$

Binders

- Prefix **in**$(\ldots, !x\ , !l\ , !X\ , \ldots)@\ell.P$ 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 "$\|$", i.e.

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

$$
\begin{array}{rrcll}
(\textsc{Alpha}) & N & \equiv & N' & \text{if } N =_\alpha N' \\
(\textsc{RCom}) & (\nu s_1)(\nu s_2)N & \equiv & (\nu s_2)(\nu s_1)N & \\
(\textsc{Ext}) & N_1 \parallel (\nu s)N_2 & \equiv & (\nu s)(N_1 \parallel N_2) & \text{if } s \notin fl(N_1) \\
(\textsc{Abs}) & s ::_\rho C & \equiv & s ::_\rho (C \,|\, \mathbf{nil}) & \\
(\textsc{Clone}) & s ::_\rho C_1 | C_2 & \equiv & s ::_\rho C_1 \parallel s ::_\rho C_2 & \\
(\textsc{PDef}) & s ::_\rho A(\bar{p}) & \equiv & s ::_\rho P[\bar{p}/\bar{f}] & \text{if } A(\bar{f}) \triangleq P
\end{array}
$$

KLAIM: Operational Semantics (1)

Producing data:

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

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'

  $\rightarrow$ *static binding discipline*

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

KLAIM: Operational Semantics (2)

Consuming data:

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

where:

- function $match(T, t)$ checks compliance of $\mathcal{E}[\![\, T \,]\!]_\rho$ 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

$$match(!x, v) = [{}^{v}\!/\!{}_{x}] \qquad match(v, v) = \epsilon$$

$$match(!l, s) = [{}^{s}\!/\!{}_{l}] \qquad match(s, s) = \epsilon$$

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

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

KLAIM: Operational Semantics (3)

Accessing data:

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

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

KLAIM: Operational Semantics (4)

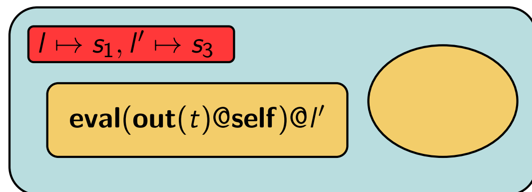
Spawning new processes:

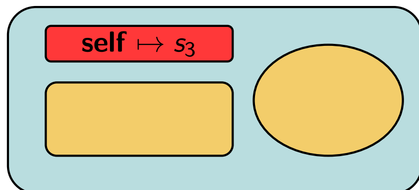
$$\frac{\rho(\ell) = s'}{s ::_{\rho} \textbf{eval}(Q)@\ell.P \;\parallel\; s' ::_{\rho'} \textbf{nil} \;\longmapsto\; s ::_{\rho} P \parallel s' ::_{\rho'} Q}$$

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

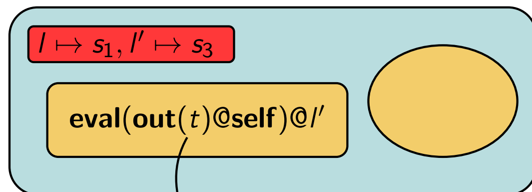
$\rightarrow$ *dynamic binding discipline*

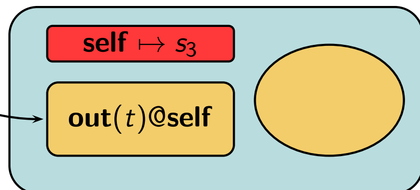
Dynamic Scoping



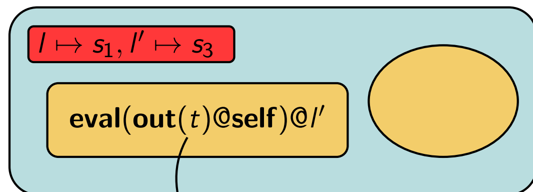


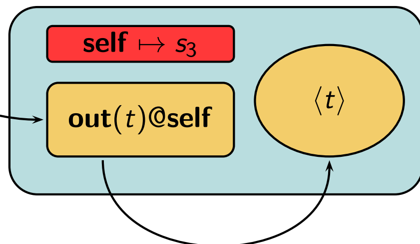
Dynamic Scoping



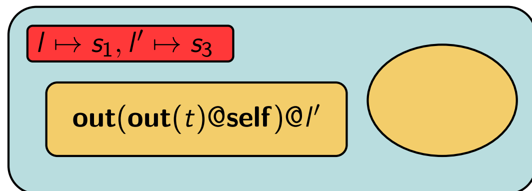


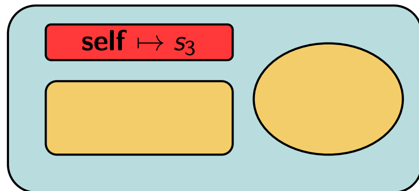
Dynamic Scoping



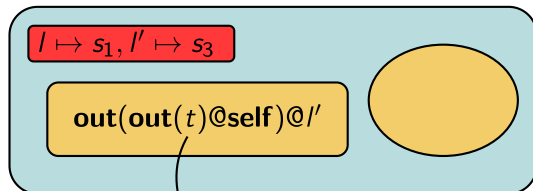


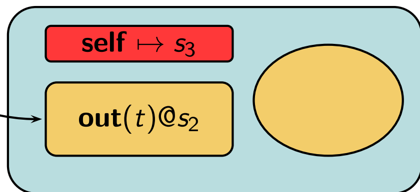
Static Scoping



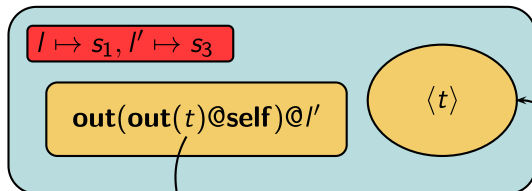


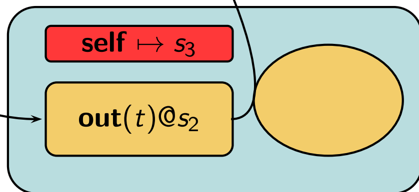
Static Scoping





Static Scoping





Klaim: Operational Semantics (5)

Creating new nodes:

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

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

Klaim: Operational Semantics (7)

Rules summary:

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

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

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

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

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

KLAIM: Operational Semantics (6)

Remaining rules:

$$\frac{N_1 \;\longmapsto\; N_1'}{N_1 \parallel N_2 \;\longmapsto\; N_1' \parallel N_2}$$

$$\frac{N \;\longmapsto\; N'}{(\nu s)N \;\longmapsto\; (\nu s)N'}$$

$$\frac{N \;\equiv\; M \;\longmapsto\; M' \;\equiv\; N'}{N \;\longmapsto\; N'}$$

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 $s_0, \ldots, s_{n-1}$
  - 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 $s_{i+1 \ mod \ n}$
- At the outset, each process retrieves an *id* from the node *rg* that acts as a random generator
  - The *id*s 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 *id*s
  - 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

$$
\begin{array}{lcl}
P & \triangleq & \mathbf{in}(\text{``}ID\text{''}, !\,id)@rg. \\
 & & \mathbf{out}(\text{``}ID\text{''}, id)@\mathbf{self}. \\
 & & \mathbf{eval}(checker(id))@next.\mathbf{nil}
\end{array}
$$

$$
\begin{array}{lcl}
checker(myId) & \triangleq & \mathbf{read}(\text{``}ID\text{''}, !\,x)@\mathbf{self}. \\
 & & \mathbf{if}\ myId < x\ \mathbf{then} \\
 & & \quad \mathbf{eval}(checker(myId))@next.\mathbf{nil} \\
 & & \quad \mathbf{else\ if}\ myId > x\ \mathbf{then} \\
 & & \qquad\qquad \mathbf{eval}(notifier(myId))@next.\mathbf{nil} \\
 & & \qquad\qquad \mathbf{else\ out}(\text{``}LEADER\text{''})@\mathbf{self}.\mathbf{nil}
\end{array}
$$

$$
\begin{array}{lcl}
notifier(myId) & \triangleq & \mathbf{read}(\text{``}ID\text{''}, !\,x)@\mathbf{self}. \\
 & & \mathbf{if}\ x = myId\ \mathbf{then} \\
 & & \quad \mathbf{out}(\text{``}FOLLOWER\text{''})@\mathbf{self}.\mathbf{nil} \\
 & & \quad \mathbf{else\ eval}(notifier(myId))@next.\mathbf{nil}
\end{array}
$$

(Mild) Extensions

By exploiting pattern matching and parallel composition,
the conditional construct

$$\textbf{if } \mathit{bcond} \quad \textbf{then } P \quad \textbf{else } Q$$

can be rendered as

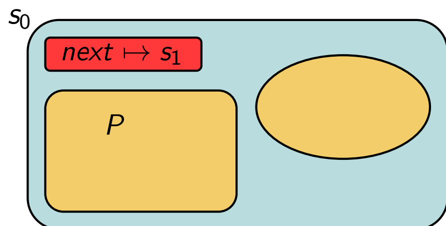
$$\begin{array}{l}\textbf{out}(\text{"}\mathit{EVALUATE}\text{"}, \mathit{bcond})@\textbf{self}. \\ \quad (\textbf{in}(\text{"}\mathit{EVALUATE}\text{"}, \mathit{true})@\textbf{self}.P \\ \quad \mid \\ \quad \ \textbf{in}(\text{"}\mathit{EVALUATE}\text{"}, \mathit{false})@\textbf{self}.Q)\end{array}$$

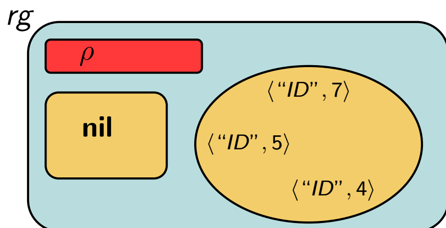
since only one of the two parallel processes can proceed

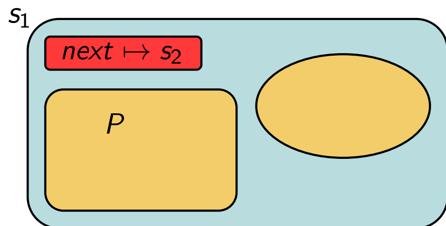
Leader Election: specification

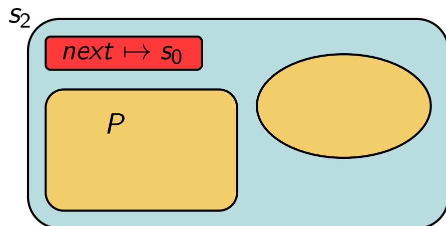
$$s_0 ::_{[\mathbf{self} \mapsto s_0, next \mapsto s_1]} P$$
$$\|$$
$$s_1 ::_{[\mathbf{self} \mapsto s_1, next \mapsto s_2]} P$$
$$\|$$
$$s_2 ::_{[\mathbf{self} \mapsto s_2, next \mapsto s_0]} P$$
$$\|$$
$$rg ::_{[\mathbf{self} \mapsto rg]} \langle \text{“}ID\text{”}, 7 \rangle \mid \langle \text{“}ID\text{”}, 4 \rangle \mid \langle \text{“}ID\text{”}, 5 \rangle \mid \mathbf{nil}$$

The overall system

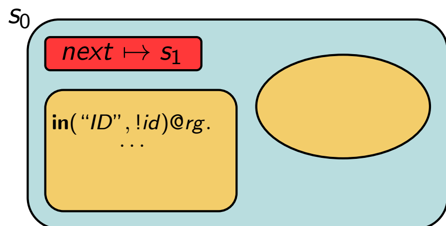


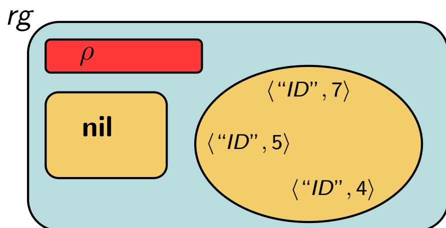


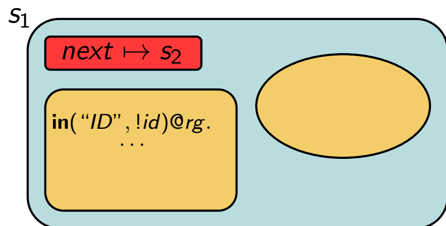


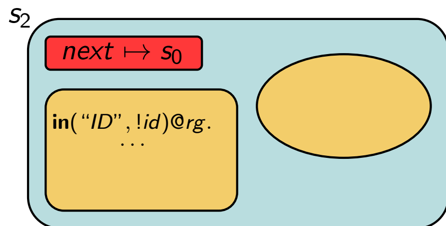


A computation

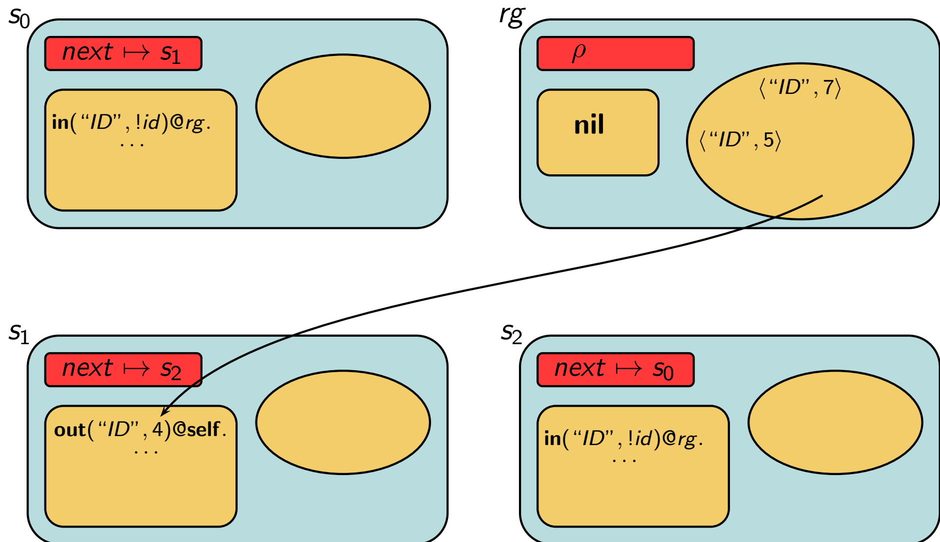




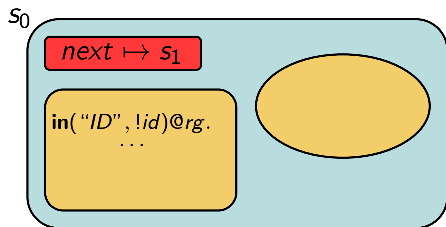


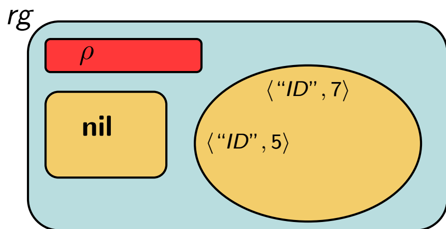


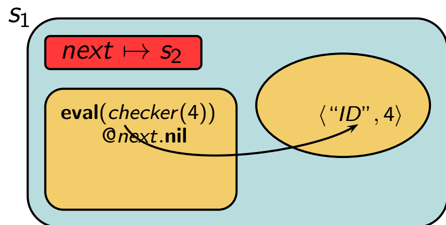
A computation

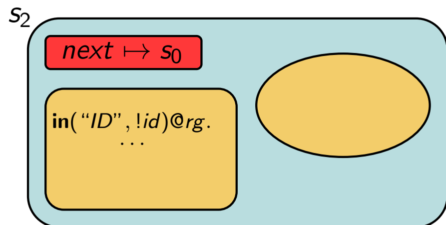


A computation

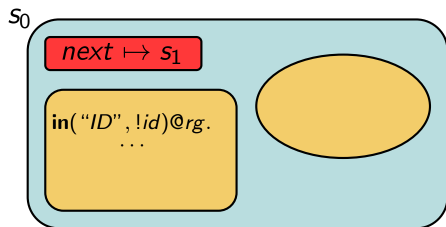


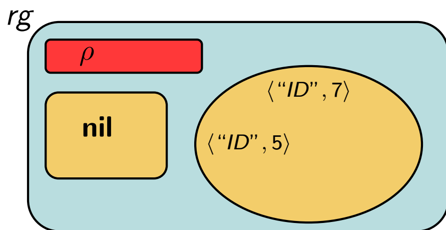


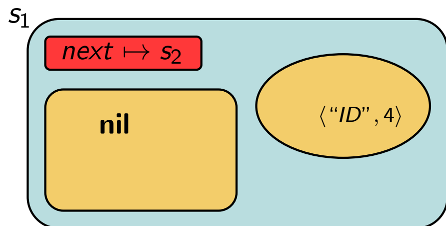


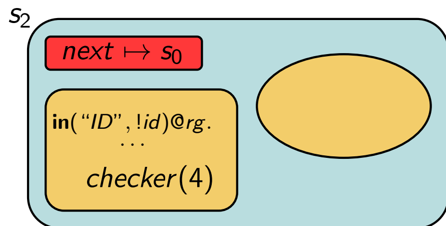


A computation

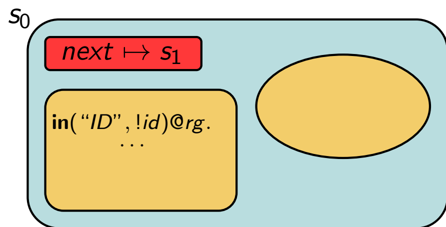


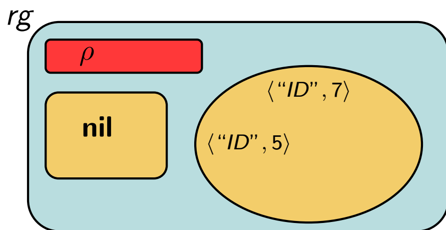


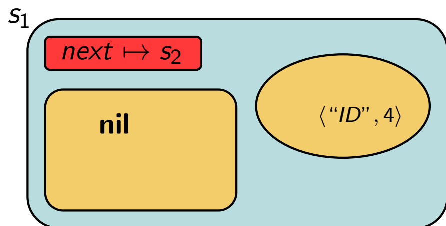


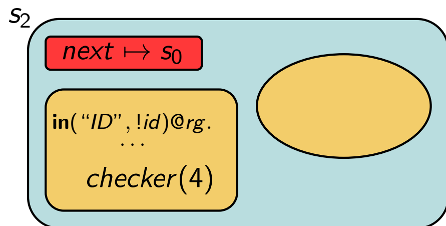


A computation

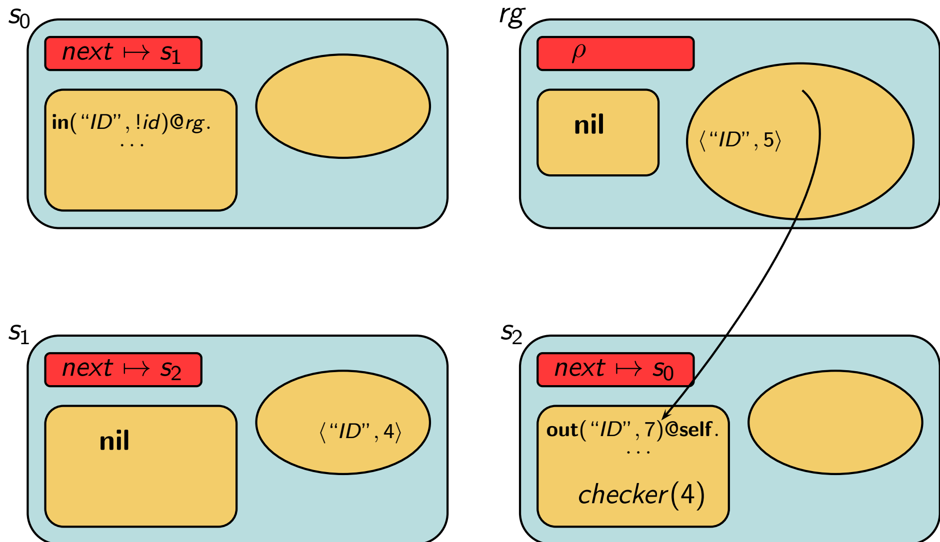




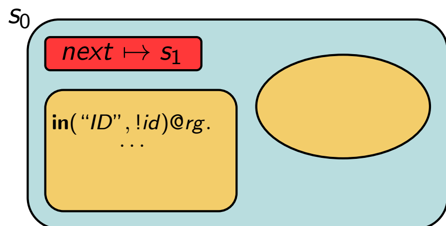


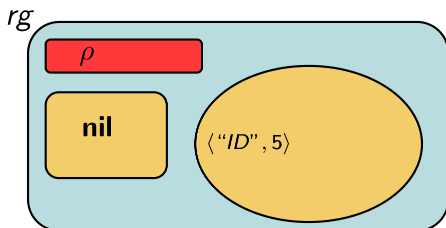


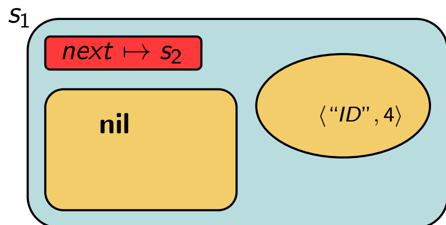
A computation

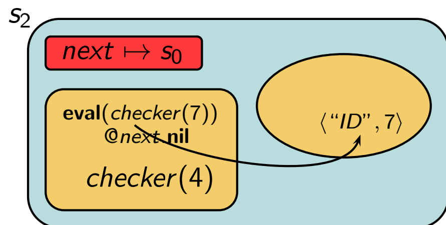


A computation

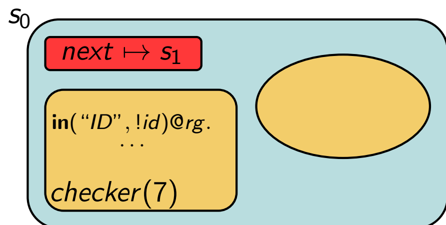


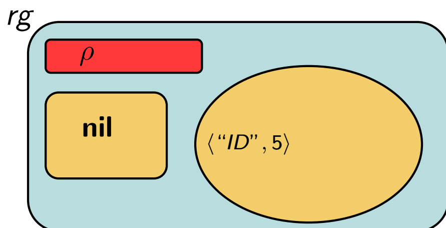


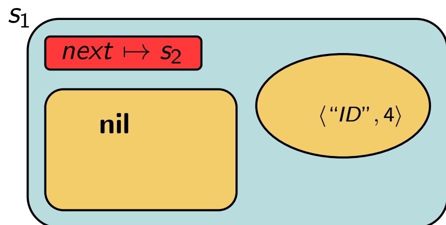


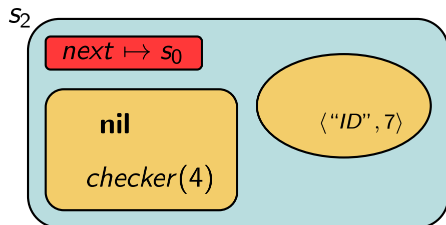


A computation

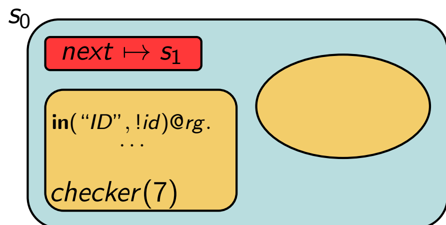


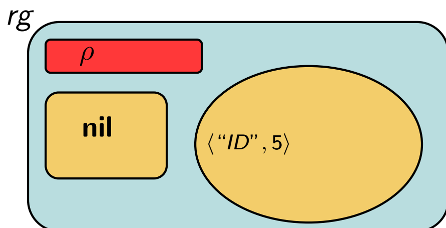


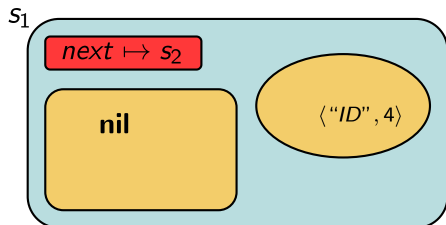


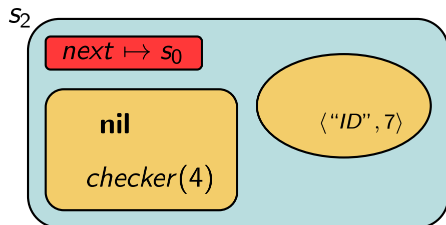


A computation

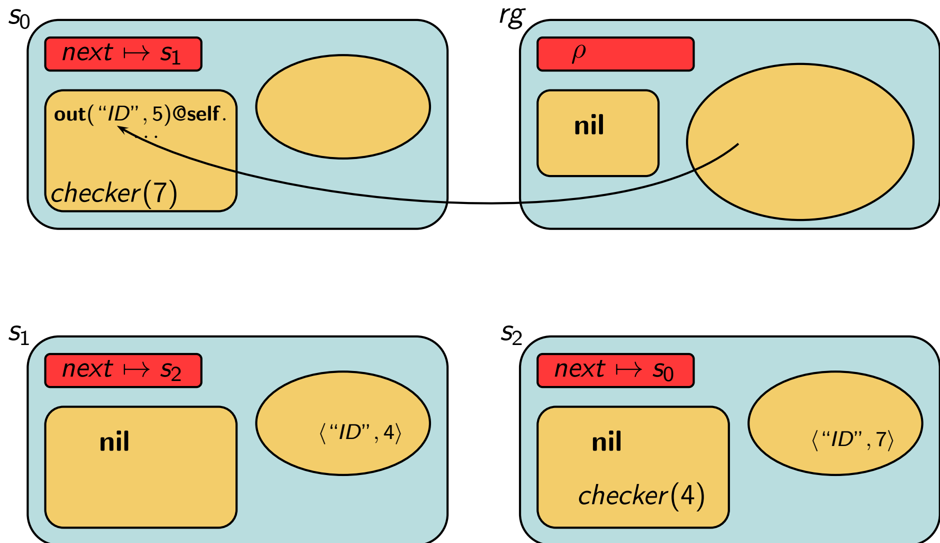




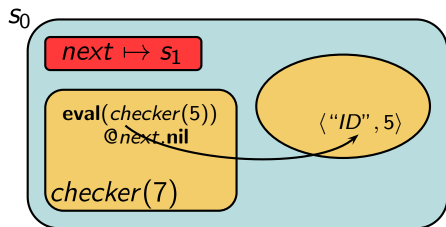


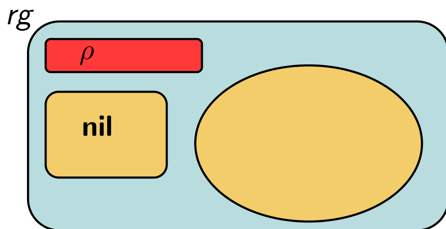


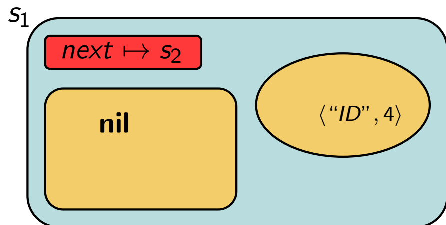
A computation

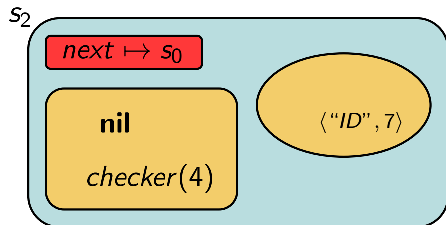


A computation

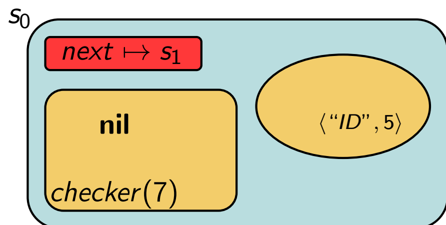


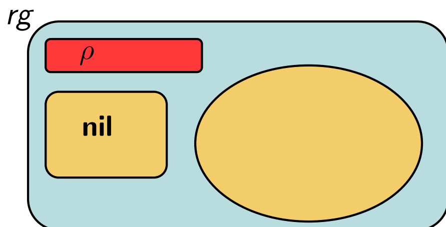


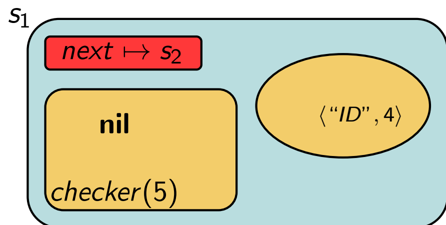


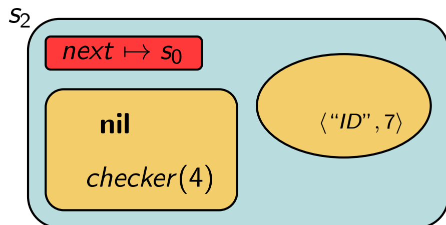


A computation

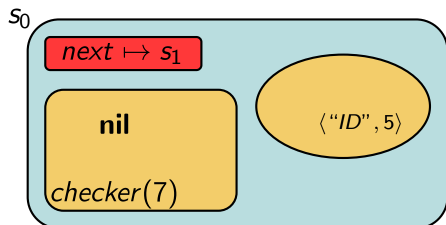


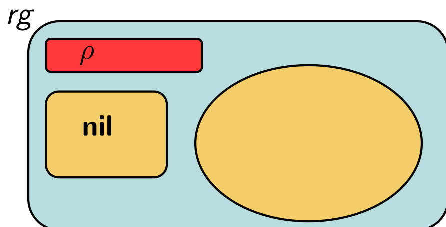


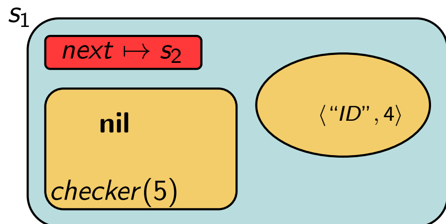


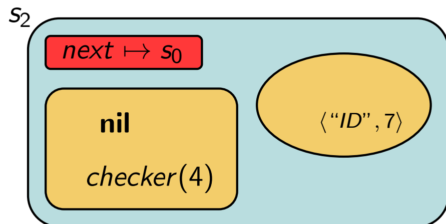


A computation

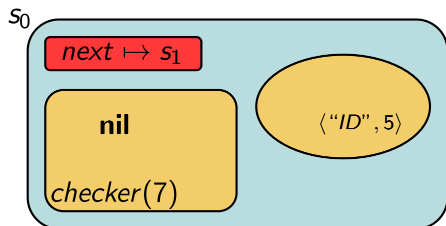


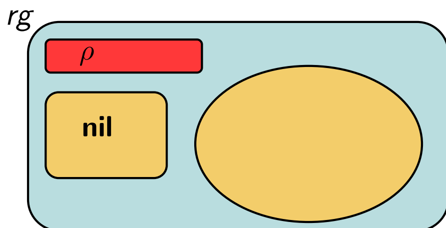


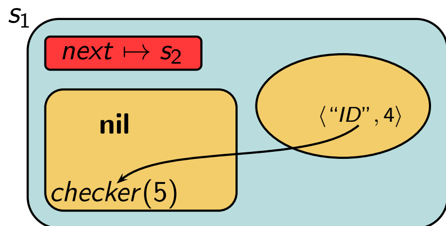


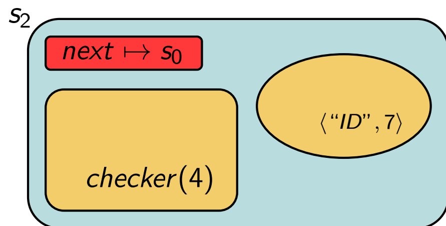


A computation

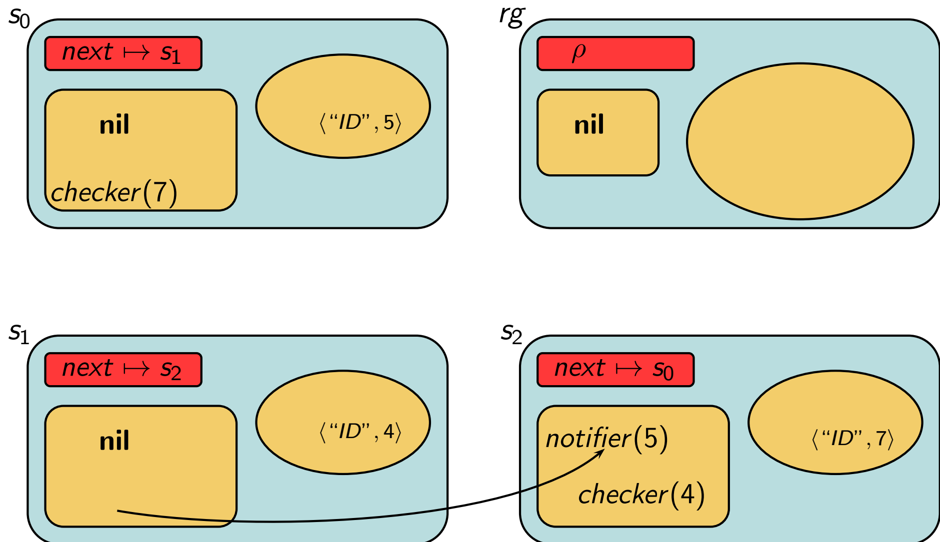




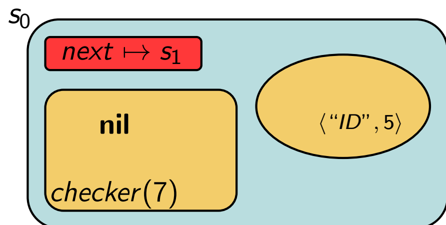


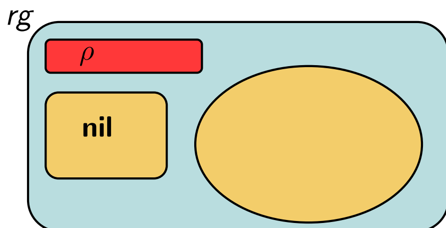


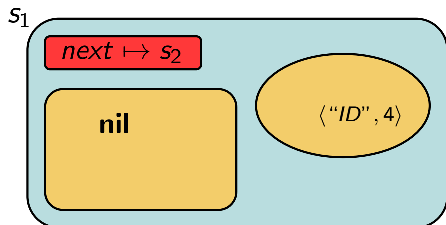
A computation

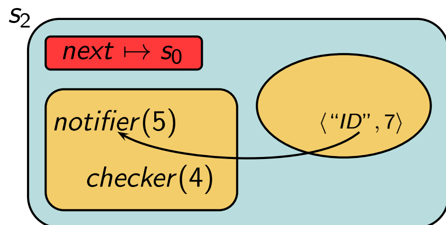


A computation

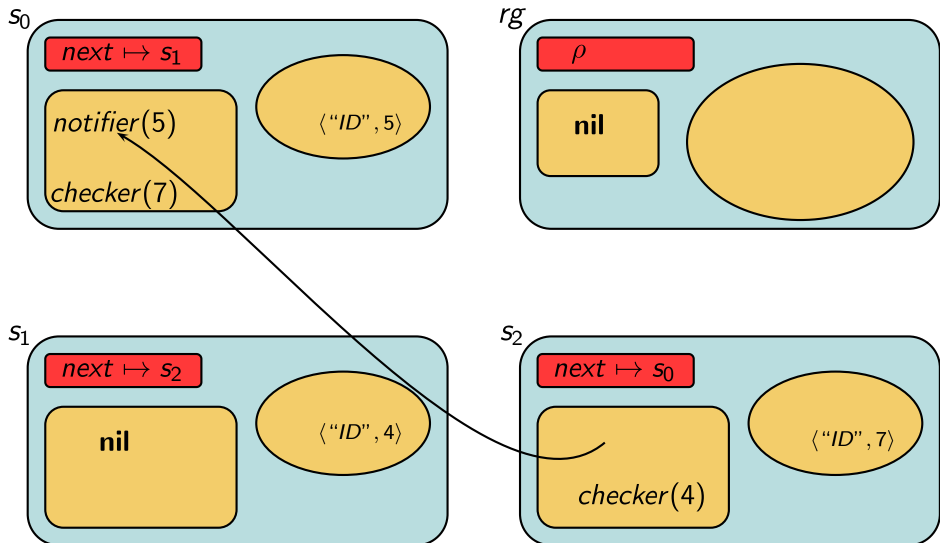




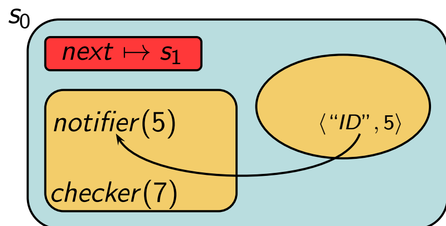


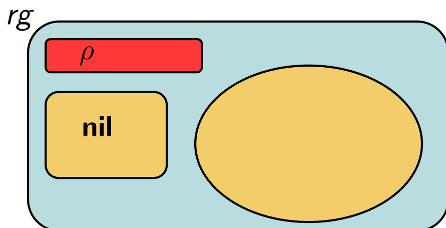


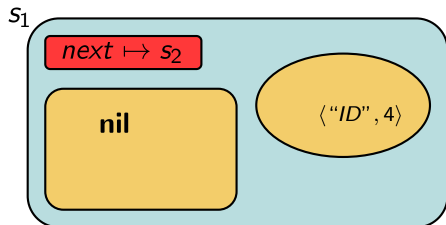
A computation

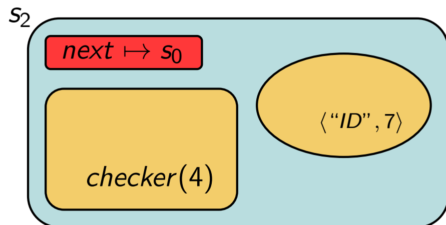


A computation

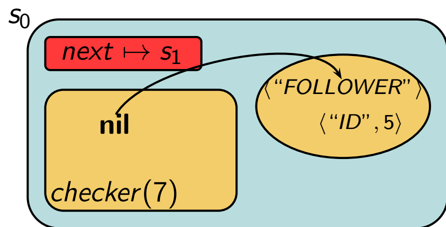


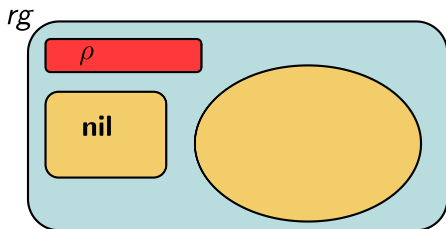


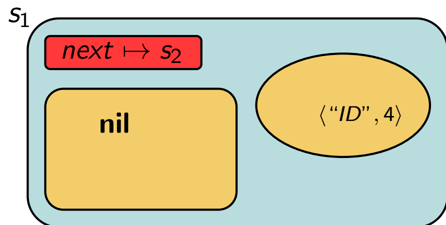


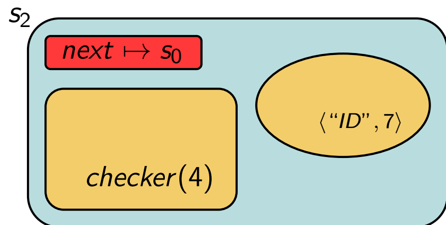


A computation

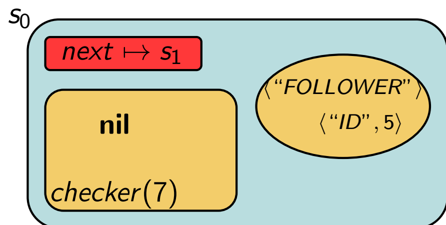


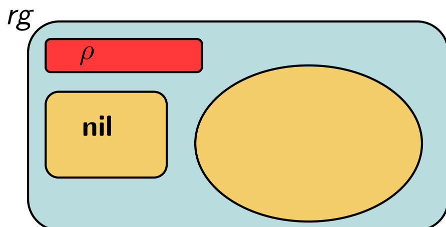


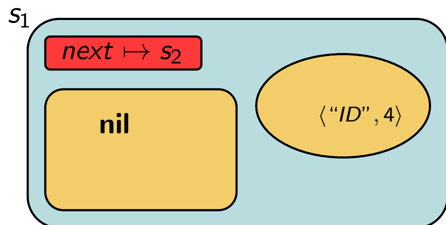


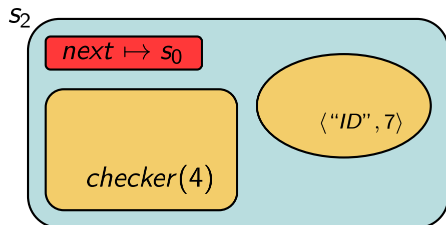


A computation

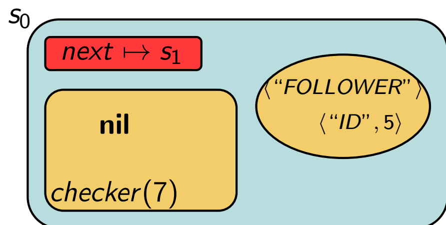


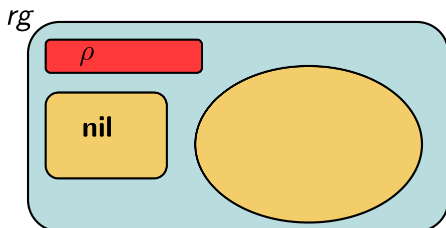


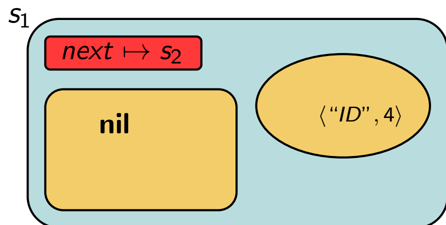


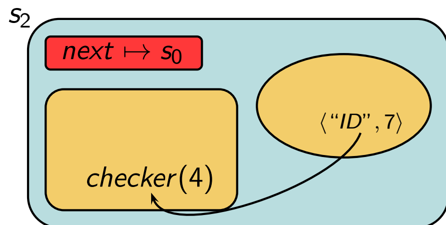


A computation

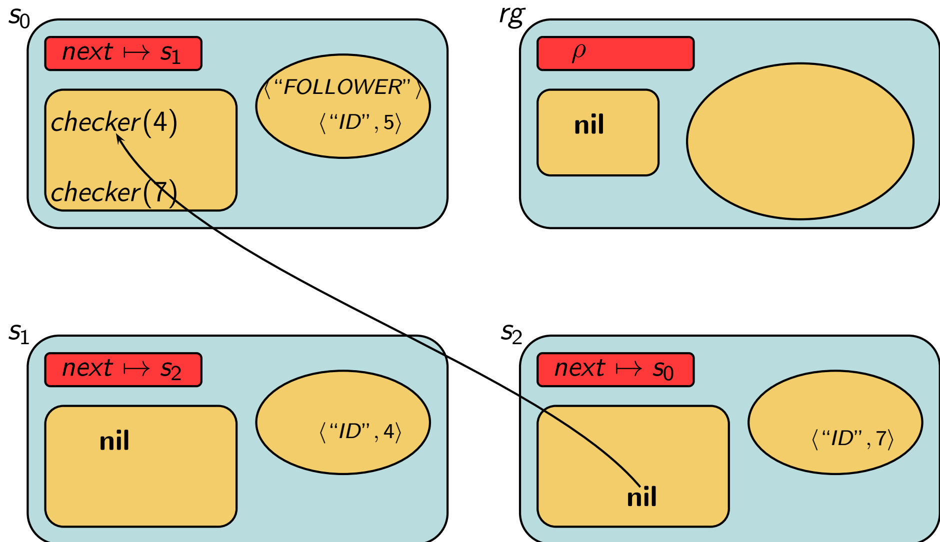




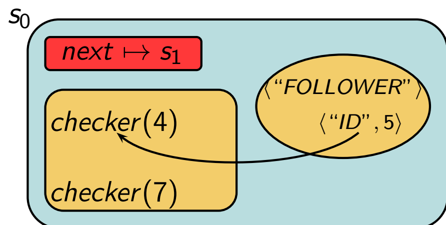


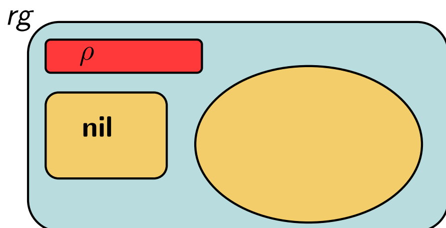


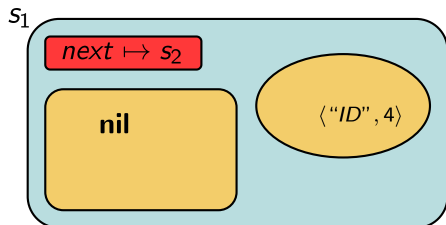
A computation

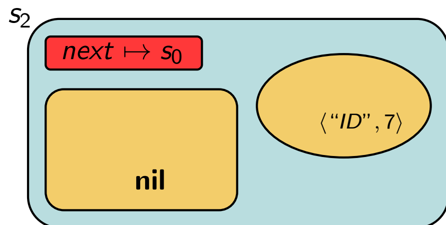


A computation

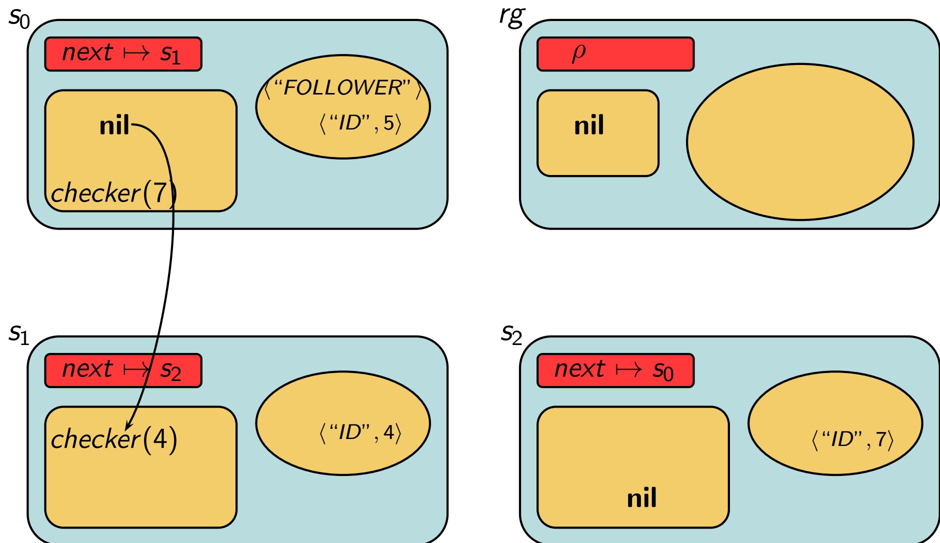




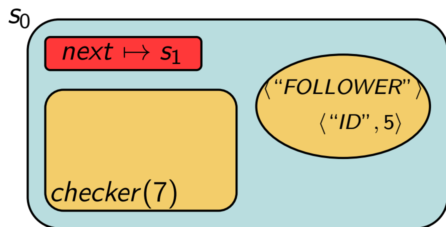


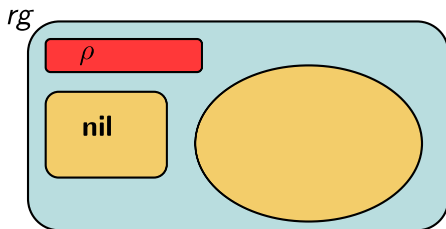


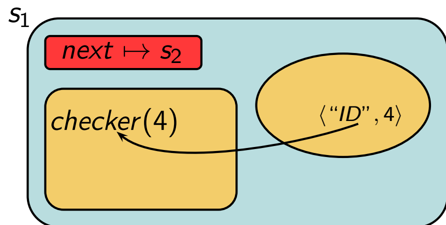
A computation

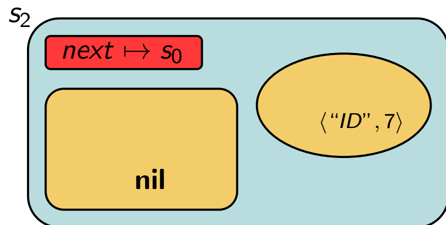


A computation

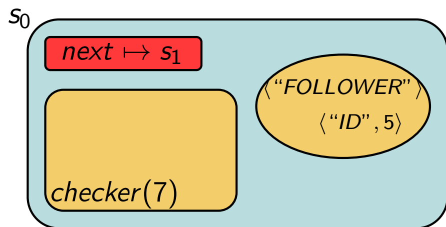


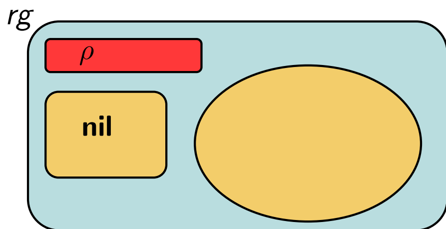


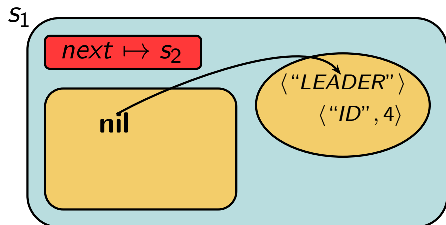


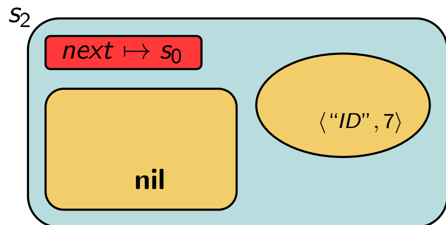


A computation

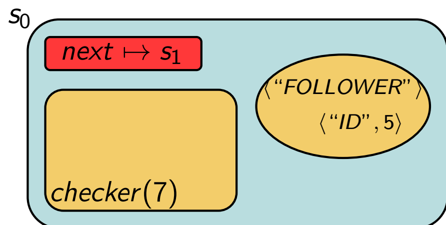


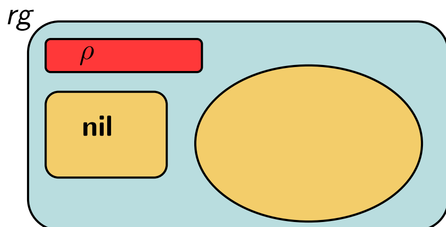


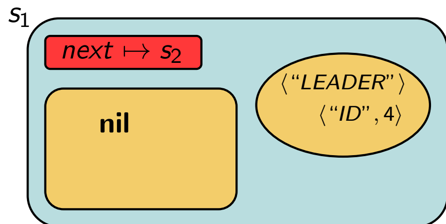


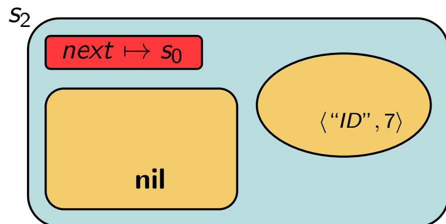


A computation

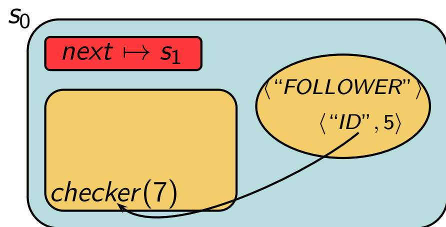


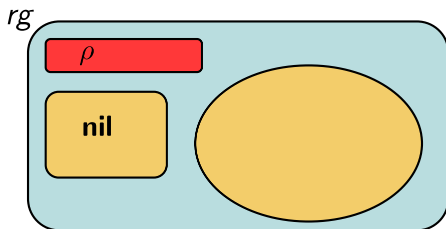


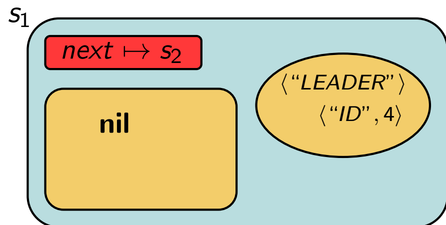


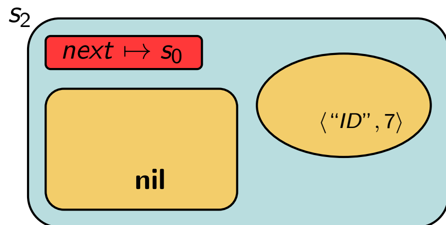


A computation

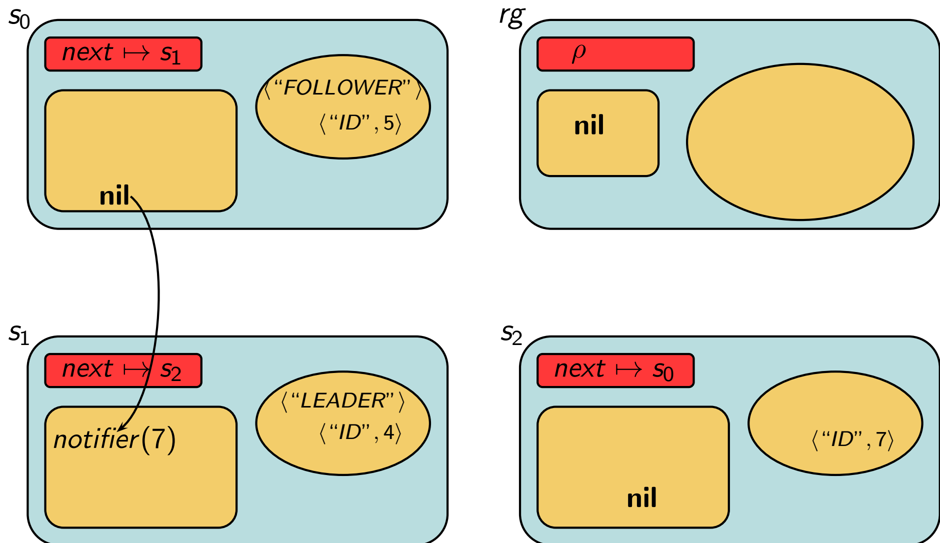




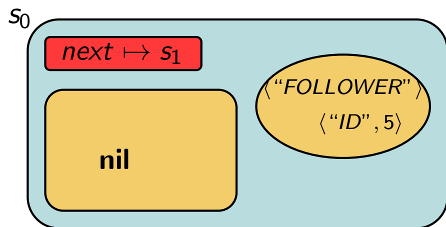


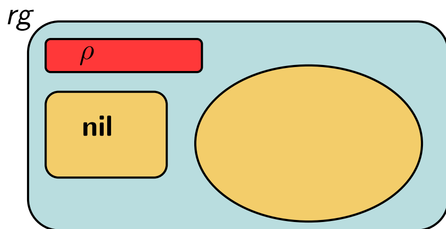


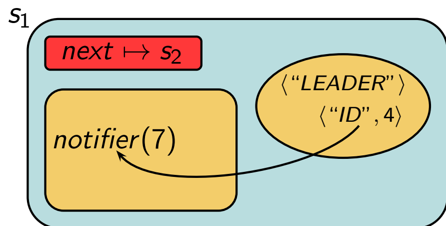
A computation

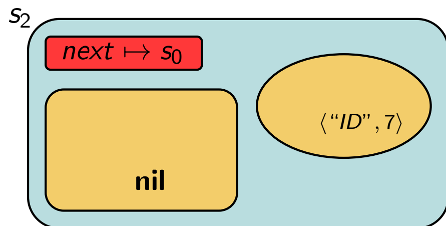


A computation

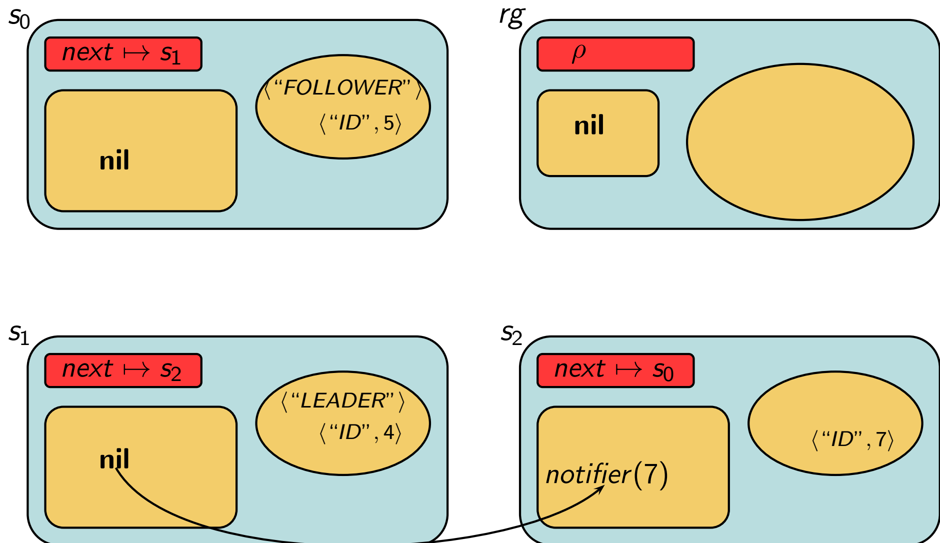




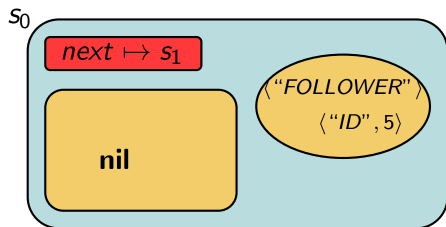


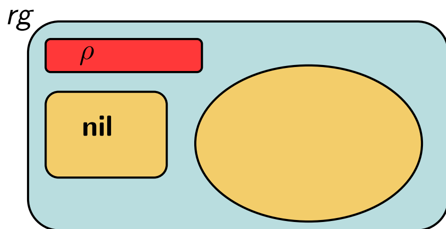


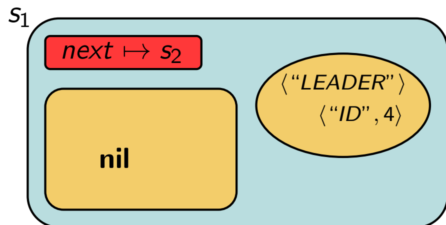
A computation

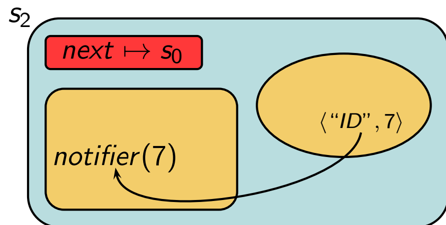


A computation

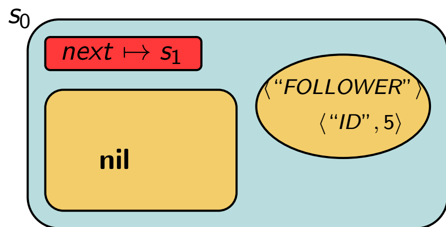


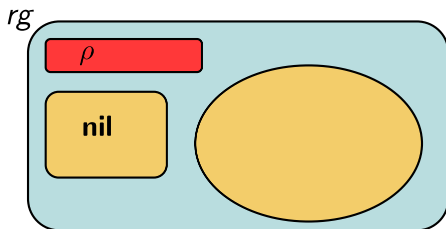


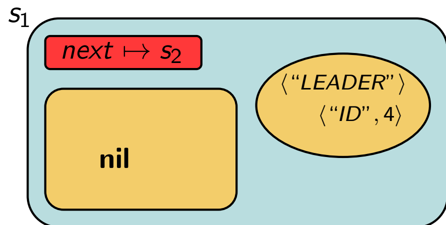


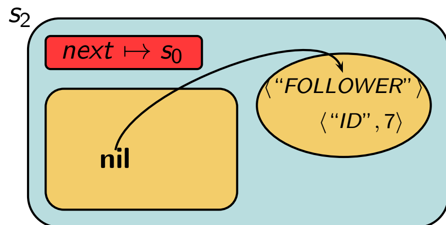


A computation

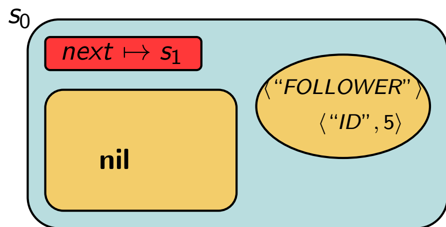


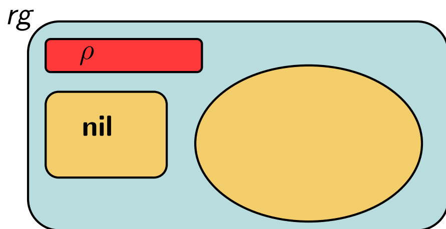


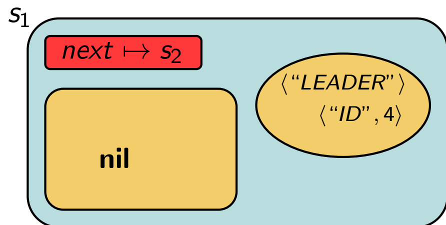


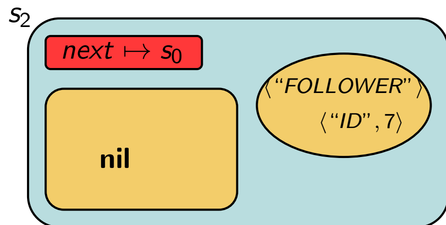


A computation









Implementation

X-KLAIM (eXtended KLAIM)

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

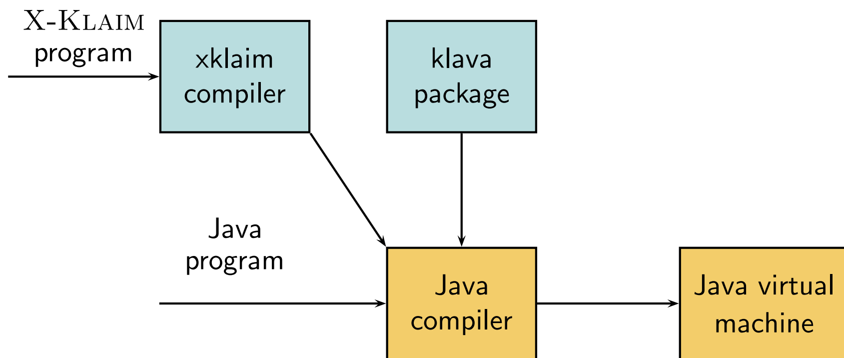
- Variable declarations
- Assignments
- Conditionals, iteration, ...
- Time-outs

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